Broadview®
www.broadview.com.cn

一站式了解深度学习算法，结合实际工作快速上手！

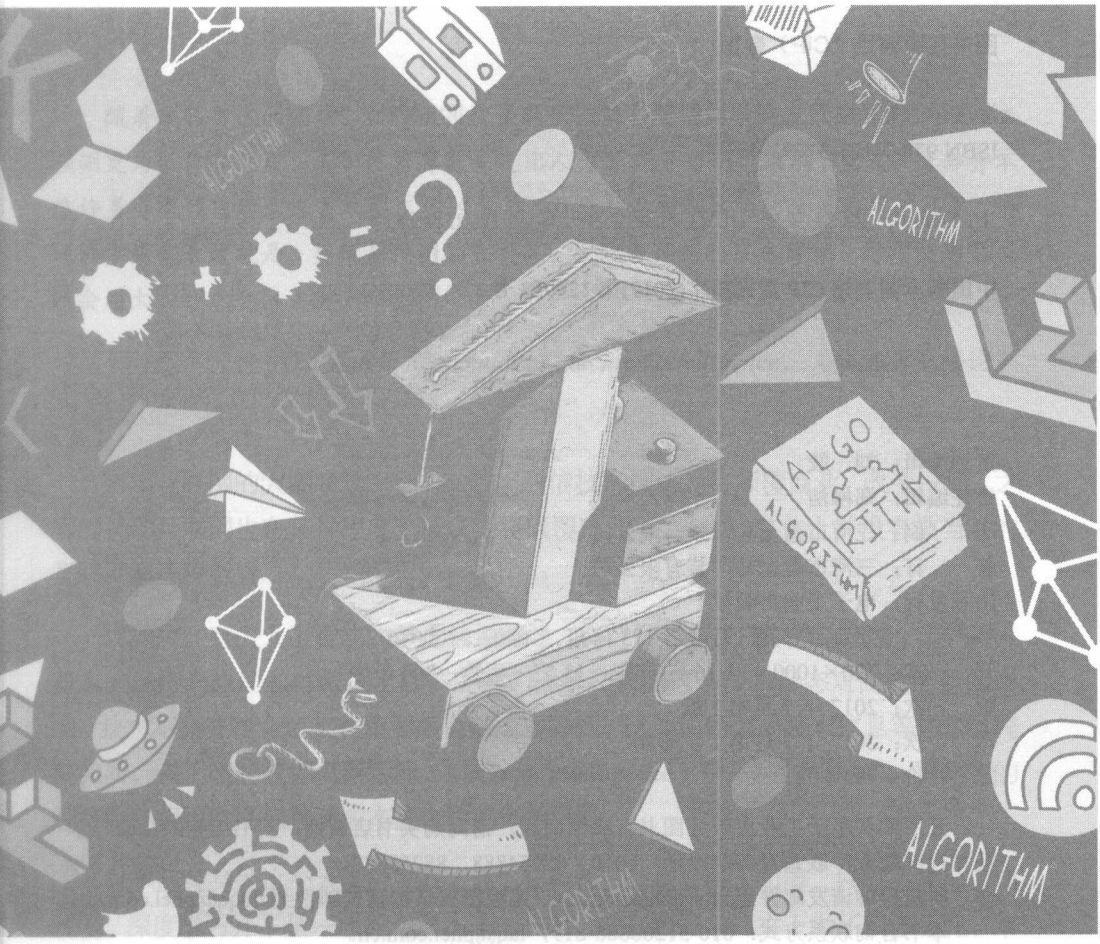
深度学习算法实践

吴岸城　编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

深度学习算法实践

吴岸城　编著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书以一位软件工程师在工作中遇到的问题为主线，阐述了如何从软件工程思维向算法思维转变、如何将任务分解成算法问题，并结合程序员在工作中经常面临的产品需求，详细阐述了应该怎样从算法的角度看待、分解需求，并结合经典的任务对深度学习算法做了清晰的分析。

本书在表达上深入浅出，让有志于学习深度学习的读者，能够快速地理解核心所在，并顺利上手实践。

图书在版编目（CIP）数据

深度学习算法实践 / 吴岸城编著. —北京：电子工业出版社，2017.7
ISBN 978-7-121-31793-4

Ⅰ. ①深… Ⅱ. ①吴… Ⅲ. ①机器学习－算法 Ⅳ.①TP181

中国版本图书馆 CIP 数据核字(2017)第 129958 号

策划编辑：刘　皎
责任编辑：郑柳洁
印　　刷：三河市鑫金马印装有限公司
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：720×1000　1/16　印张：13.5　字数：204 千字
版　　次：2017 年 7 月第 1 版
印　　次：2017 年 7 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前　言

随着机器智能的进步，人类预测技能的价值将会降低。原因在于机器预测比人工预测更为低价和优质，正如机器算数肯定比人力算数更为迅速准确。然而，这却并不像许多专家预言的意味着人类工作的末日，因为人类判断技能的价值将得以凸显。用经济学语言表述就是，判断是预测的互补品，因此当预测的成本降低，对判断的需求就会增大。

——*The Simple Economics of Machine Intelligence*，《哈佛商业评论》

当年互联网的大潮席卷一切时，数字通信技术被认为将颠覆商业、改变一切。之后的移动互联网也在某种程度上被认为将颠覆商业。经济学家总体上并没有被当时的互联网泡沫所忽悠。

现如今，有关人工智能的报道铺天盖地，有昔日“新经济”泡沫之势。这一次，基本的经济学原理和框架就足以帮助我们理解和预测这一技术形态对商业产生的影响。技术革命往往会使某些重要活动的成本降低，比如说通信或搜索信息等活动。究其实质，人工智能或者机器智能（Machine Intelligence）是一项预测技术，因此它的经济影响将围绕降低预测成本这个中心来展开。

对于已经身处这个大潮中的开发者、架构师、数据分析人员等，只能去拥抱这项技术。深度学习并不是一项凭空冒出来的技术，它在机器学习之上做了很多优化。本

质上所有的有监督学习都是在探讨怎样无限地逼近目标函数（强化学习另外讨论），而深度网络就是让机器代替人类提取特征的工作变得更有可能真正实现。在经济学家的眼里，现阶段人工智能的本质是从预测（或分类问题）开始，我想通过几个实际的例子来和大家聊聊这个话题；另一方面，我的团队在工作中积累了一些实际经验，我们也希望能将这些经验贡献出来，如果能在某种程度上对读者有所帮助那就最好不过了。以上两点，促成了本书的诞生。

我写的前一本书（《神经网络与深度学习》，电子工业出版社出版）偏向于概念讲解，因为写的时候深度学习并不普及，让大众了解深度学习的基本概念是最急迫的目标。本书大部分内容则偏向于应用，因为无论是降低社会成本，还是发明创造出新算法，都离不开实践；如果还能从实践中思考一些东西，那就是举一反三的能力了——这是我们人类独特的价值。所以在本书后面的强化学习、股票预测等章节，我们都会留一些问题，读者可以亲自实践，用深度学习这个工具创造出更多的价值，更长远地说，为推动强人工智能贡献自己的一份力量。

我一直相信，创造具有意识的 AI，对所有的科技人员都是一种诱惑，尽管有可能造出来就意味着人类的边缘化，但仍然要憋着劲儿去研究如何把它造出来——这简直就不像人类的自由意志，更像背后有一只手在推动着，我想上帝在造人时的心情也不过如此吧。

本书面向有一定基础、在工作中对深度学习有一定实际需求的读者；也面向那些有志于从传统的软件工程领域转型的工程师们。

本书一共分为 6 章。

第 1 章，主要讲从工程思维到算法思维的转变，对于有基础的读者来说稍显啰唆，但很重要，希望读者能仔细阅读。

第 2 章，阐述文本分析、文本深度特征等内容，已有基础的读者可以根据自己的需求部分略过。

第 3 章，主要介绍对话机器人的相关技术和发展。

第 4 章，主要介绍视觉，以人脸检测为例，从传统的 OpenCV 模式识别做人脸检测到用 CNN 网络做人脸表情识别。勾勒 CNN 的传承发展，讲述做图像分类、目标

识别等其他应用。

第 5 章，主要讲区别于一般的有监督学习的另一个问题：强化学习和 DQN 网络实践。

第 6 章，主要讲预测与推荐，以股票为例，并同时讨论了深度学习在推荐领域的应用。

本书的完成得到了团队的大力支持：张帅、郭晓璐提供了图像方面的支持；周维提供了强化学习内容上的支持，在此衷心地感谢他们。

下面是本书用到的环境说明。

为保持一致性，本书所有的代码如无特殊说明都基于 Python 2.7 版本，系统环境为 Ubuntu 16.04，以及所用到的 Keras、TensorFlow、MXNet、Caffe、Openface、openAI、OpenCV、Dlib、NumPy、SciPy、Gensim、Theano 等均为 2016 年 9 月的最新版本。

本书的部分源代码整理在：https://github.com/wac81/Book_DeepLearning_Practice，仅供研究使用，使用时请注明来源，如需用作商业或其他用途，请联系作者取得授权。

目　录

开始

1.1 从传统的软件工程思维转型

2013 年的秋天，这时英特已经从中国一流学府的计算机系毕业 3 年了，他所在的这家公司早已在纳斯达克上市，而英特已经成为这个公司的中层。这 3 年来，英特初出茅庐的那些热情已经被磨砺得差不多了，但他的心底还是在隐隐渴望着什么。对如今的工作，他总觉得缺少些激情，总觉得有点按部就班。从当年的研发工程师到如今的研发管理中层的位置，这种感觉一直如影随形。

今天英特被总经理叫到办公室讨论部门以后的规划，老板看最近大数据已经开始火了，想要英特组建一个大数据的团队，这个团队主要是通过对公司数据仓库的迁移和计算，看看能不能打造出什么新产品。

英特接手此事后，便开始调研，发现如今只要内部开始做大数据的公司，都无一例外地弄了一个算法团队。然而这些算法团队基本上对后期的想法和规划很不清晰，感觉大家都是在摸着石头过河。于是，英特也想配套着弄一个算法团队。

略去大数据团队不说，单说算法团队，英特的公司里就没有这样的人才。这意味着英特需要开始招兵买马了。英特先从一家知名电商公司挖来了一名高级算法人才，

然后照着行业内其他一些公司的做法，组建了相关的数据标注员，并在公司内部挑选出了一些有经验的 Python、C++工程师。

现在英特面临着一个问题：大部分人员都是做工程的研发工程师，现在他们的目标是用编程语言实现算法，那么实现算法与之前做一个工程项目有区别吗？如果还按照之前那样，从工程的角度思考问题，能否实现效果或者能否达到业务部门的要求呢？

英特开始时很乐观，正好这时公司的业务部门给算法团队提了第一个需求——“先把垃圾邮件给过滤下吧，最好对于正式的邮件也做一些分类，比如根据文本内容推荐重要或紧急邮件，对于一般邮件不做重点过滤。”英特接到这个需求后，马上做了些调研，发现这个需求就是一个分类需求，用任何一个分类器都可以比较快地搞定。

那英特如何做呢？按照现有的软件工程的思维，是需求分析→逻辑设计→详细设计→测试→上线，整个过程英特非常熟悉。然而按照这种过程，算法团队的小朋友们还没有干到逻辑设计这一步，就已经被业务部门催促了，“我们只要一个功能你们动作怎么这么慢”。在两个月的煎熬之后，英特总算给出了第一个可以使用的模型，用在对重要邮件、垃圾邮件的分类中。但仅仅过去一天，英特的电话就被打爆了，大家几乎都在抱怨说，重要邮件被分到垃圾邮件中，非重要邮件被推荐了。英特感到很委屈，试图去解释，却又被业务部门讽刺了一顿。重新梳理了整个流程之后，英特得出一个结论：对于算法而言应该有算法的思维。不能完全套用以前做软件工程的那套东西。

1.2 建立算法思维

看看业务部门抱怨的两点：第一，算法团队反应太慢。第二，没有达到预期效果。从中我们隐约可以感觉到传统的软件工程思维并不适合做算法。

下面，我们就和英特一起来梳理下为什么传统的软件工程思维不适合做算法，以及算法的开发流程和步骤应该是怎样的？

1.2.1 算法的开发流程

用算法解决项目中的问题实际上是一种实验思维。实验思维顾名思义，就是我们用初始数据做实验，输出一些实验产物，对于算法来说就是一些向量或矩阵，也就是一堆数字，我们得到这些数字后可以开始反馈到现实业务中，用于比较输出值和真正的业务需求是不是具有一致性。所以做算法的开发流程完全可以抽象为如图 1-1 所示的框架图。

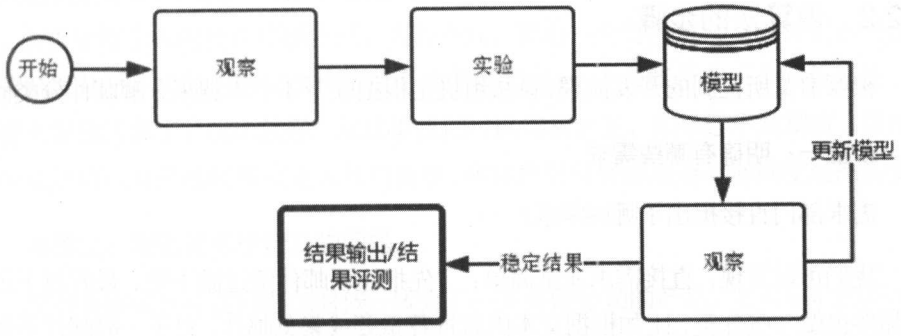


图 1-1 算法开发流程

举个例子，我们开发一个算法时，一般是要用来替代人的一部分工作的，所以我们会先了解人是怎么完成这部分工作的。比如邮件分类，就先来看看人是如何做邮件分类的。

（1）看邮件的标题，很多信息其实在标题上就有呈现；

（2）注意发件人，看用户名是不是一堆无意义的字符；

（3）看邮件内容，邮件内容不用看完，优先去找到一些重点，对于中文的垃圾/重要邮件，会看看有没有“发票”、“讲座”、“小姐”之类的词；

（4）如果邮件带附件，要对附件名进行仔细的甄别。

了解人是怎么来完成邮件分类的，我们就完成了邮件分类算法的前期观察。接着我们将观测到的重要特征抽象出来，这个时候可以人工选择特征，也可以用机器学习的方法抽象一些特征。针对邮件分类问题，我们可以用特征做分类算法，可以采取的分类算法有很多，你既可以选用普通的机器学习算法，比如贝叶斯、SVM、随机森

林等，也可以选用神经网络来进行分类。最后我们将结果回馈到我们观察的事物中去，简单地说，这一步是一个验证和调整的过程，这一步的好坏也直接决定了我们最后的结果是好是坏，这其实也是区分一个有经验的算法工程师和一个初级的算法工程师的标准。

刚才我们简单地了解了算法的开发流程，下面我们再来看看英特是如何做好观察并一步一步完成需求的。

1.2.2 做算法的步骤

根据前文所提到的开发流程，算法组决定根据以下 4 个步骤来分解邮件分类问题。

步骤一：明确有哪些需求

需求部门直接提出了哪些需求？

我们可以发现，直接需求非常简单："先把垃圾邮件给过滤下吧，最好对于正式的邮件也做一些分类，比如根据文本内容推荐重要或紧急邮件，对于一般邮件不做重点过滤。"

通过分解直接需求可以得出两个需求：

（1）过滤垃圾邮件。

（2）分类非垃圾邮件，建议分为重要紧急邮件和一般邮件两类。

其实一般来说，直接需求都不难满足，这里按字面意思设计两个分类器也行，或者设计一个分类器直接将邮件分为垃圾类、重要类、紧急类三类也行。

那么，除了直接需求之外，需求部门的隐含需求有哪些呢？

再次来看看英特被需求部门抱怨的两个点：

（1）慢！

（2）分类错误多！

从这里我们可以看出，需求部门的第一个期望是准确率高，第二个期望是速度能更快。

“大多数需求在提出时，对于准确率都是有要求的！”请大家记住这句话，并默认这个准确率在 95%以上。对于一些诸如图片分类或者预测问题，我们根本达不到这个准确率的时候怎么办？如果限于目前的技术水平达不到，那就尽可能多地提高准确率，去尝试多种算法和模型，因为解决大多数实际问题都不只是用一种算法或者一个模型就能完成的。

如果最后还是达不到，或许可以换种方式引导客户，比如微软之前推出过一个根据人脸识别年龄的 App。对于工业化来说，识别要求非常高，误差也只能在 ±2 岁之内，而由于每个人的生长环境不同，人种不同，要用一个普遍意义上的模型达到这种区分精度几乎不可能。所以微软将这个功能加到 App 上并赋予它一个娱乐属性，就是要大家别当真了，玩玩就好。从这里我们可以得到启发，如果达不到精度，算法工程师或许可以和产品经理或需求部门商量，在客户引导层面规避对准确度过高的要求。

步骤二：观察需求中涉及的问题

英特观察邮件分类问题的时候，体会了一下其他公司的产品，发现它们的产品也有分类错误的时候，比如企鹅公司做的邮件分类，在垃圾邮件中确实几乎 100%都是垃圾邮件，而正式邮件中偶尔会出现垃圾邮件。但这些错误却不太影响人的主观感受。这是怎么回事呢？原来人们在浏览这些正式邮件时，会自动剔除（跳过）垃圾邮件。而自家的算法组做的模型是非常严格地区分垃圾邮件，有时候把正式邮件也当成垃圾邮件处理了：这可能会导致用户错过非常重要的邮件，因此这种结果对于用户来说简直是不可接受的！这种情况表现到数据观察上，就是精确率低了，而召回率高了。

这里我们引入三个概念：准确率（Accuracy）、召回率（Recall）、精确率（Precision）。

准确率 = 提取出的正确信息条数 / 总样本条数

精确率 = 提取出的正确信息条数 / 提取出的信息条数

召回率 = 提取出的正确信息条数 / 样本中的信息条数

比如说某人的邮箱中有 1000 封正常邮件、500 封垃圾邮件和 100 封重要邮件。现在以找垃圾邮件为目标，模型一共找到 400 封垃圾邮件、5 封重要邮件和 100 封正常邮件，那么三个指标如下。

准确率（垃圾邮件）= 400/（1000+500+100）= 0.25 = 25%

一般来讲我们的准确率不应只针对一类邮件来说，应该对于所有判断正确的邮件而言，这里包含重要邮件、正常邮件，所以准确率的计算如下。

准确率=（400+5+100）/（1000+500+100）= 0.253 = 25.3%

精确率（垃圾邮件）= 400/（400+5+100）= 0.79 = 79%

召回率（垃圾邮件）= 400/500 = 0.8 = 80%

如果有一种模型能同时将精确率和召回率提高那当然是最好的，但如果只能选择其一的话，在这个邮件分类的案例中，我们应该选择精确率提高作为我们的目标。

极端情况下，如果算法组的模型只出了一个结果，而且是准确的，那么精确率就是 100%，但是召回率就很低；而如果模型把所有结果都返回，那么召回率是 100%，但是精确率就会很低。因此在不同的场合中需要自己判断希望精确率比较高还是召回率比较高。如何找到一个比较好的比例呢？如果是读者自己做实验研究，可以绘制 Precision-Recall 曲线来帮助分析（如图 1-2 所示）。

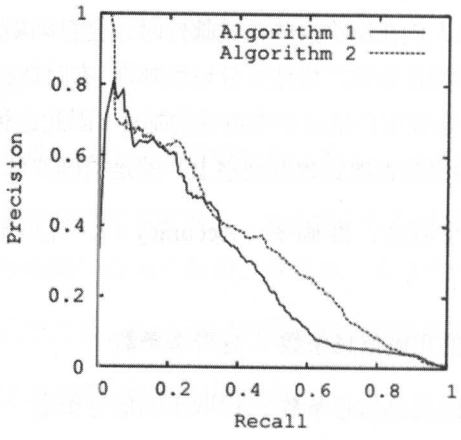


图 1-2　Precision-Recall 曲线

步骤三：开始算法工程的实践

英特在拿到需求后就开始开发算法了，但整体的进度和需求部门期望的进度还是差距很大，这是英特没有跳出传统软件思维的束缚所导致的。

对于需求部门来说，希望算法组非常快地给他们一个结果，这样就可以先试试怎么样？看看缺点在哪里？有了目标才好继续改进。而英特用软件工程的方法把算法当成了一个系统来做，这显然错误地估计了需求部门的隐含需求，也使得自己非常被动。

另外，对于算法的选用，也就是到底使用哪些算法处理业务问题，英特还是觉得有些力不从心，哪些算法可以真正带来效果的提升，光凭想象是没有意义的，一切都需要实践！

在后面的工作中，英特越来越觉得盲目地使用算法，结果往往是消耗了大量时间、精力而效果没有显著提高，对自己非常不利，也对整个算法团队不公平。那么更加务实的方法，就是去理解数据，理解用户，从分析和挖掘需求出发，先解决关键问题，再脚踏实地、一步步优化效果。对数据理解足够深刻之后，算法工程师模型方面的知识才能派上用场。对数据和业务的理解需要时间沉淀，而对于模型的灵活运用需要深厚的理论知识背景。

虽然这本书着重写的是算法，但请大家谨记下面两句话。

> 数据（业务）和算法同等重要，甚至有时候更重要。
> 不要为了用算法而用算法。

步骤四：测试

测试就是如何评价这个算法。现在英特也很头疼测试，在英特观察需求的时候，已经知道了三个用来评价模型好坏的基本参数。

但我们要重新来谈谈有关准确率、精确率和召回率这三个参数的问题。也许从最开始评价模型时，研究者们只发明了准确率一个参数，因为准确率是指对于给定的测试数据集，分类器正确分类的样本数与总样本数之比。在逻辑上这是非常清晰的，这就好比我们一个人区分垃圾邮件，找出真正的垃圾邮件占总邮件的多少，也能说明这个人寻找垃圾邮件的能力。

然后马上就发现这里有问题，我们稍微修改一下上面的数据，比如说某人的邮箱中有 1000 封正常邮件，5 封垃圾邮件，10 封重要邮件。假设现在的一个模型是判别正常邮件的，明眼人一看就知道，即使模型毫无作用，将一共 1015 封邮件都当成正

常邮件处理，这样准确率也能高达 1000/1015 = 0.985 = 98.5%，而这个“啥也没做的”所谓的模型是否真正地反映了处理能力？

看到这里，我们知道了准确率并不能真正评价一个模型的能力，所以研究者们发明了另外几个评价指标来判断，那就是精确率（Precision）、召回率（Recall）和 F_1-Measure。

这里再稍微说下 F_1，F_1 就是 Precision 和 Recall 的调和均值：

$$\frac{2}{F_1} = \frac{1}{P} + \frac{1}{R}$$

把这个式子调整一下就得到：

$$F_1 = \frac{2PR}{P+R}$$

所以我们从上面的例子来看，可以得到：

$$F_1 = 2 \times 0.79 \times 0.8/(0.79 + 0.8) = 0.794 = 79.4\%$$

F_1-Measure 默认精确率和召回率的权重是一样的，但有些场景下，我们可能认为精确率会更加重要，这时可以通过调整参数得到相应式子。

模型的测评指标有了，那如何对模型进行验证呢？有很多测试方法，比如交叉验证等，需要英特及整个算法组在后面积极思考和善加利用。

假设现在已经将模型验证完了，英特也拿到了评测数据，这个模型是否可以投入使用则需要根据这些测评指标来判断，通过不同的参数调整得到相关的测评指标，然后将这些测评指标进行比较，最后选出一个“最为平衡”的模型。注意，是“最为平衡”而不一定是某几个值最高，一切的出发点都以实际业务为准。

1.2.3 英特的总结

在整个算法项目的研发过程中，英特也注意到前人早已对算法的研发流程总结了规律。这些规律已经形成了标准，有两种方法论，一种是 SPSS[1]的 CRISP-DM，另外

1 当然现在 SPSS 已被 IBM 收购了，这里还是称为 SPSS。

一种是 SAS 搞的 SEMMA。

（1）先来看看 CRISP-DM 流程（如图 1-3 所示）。

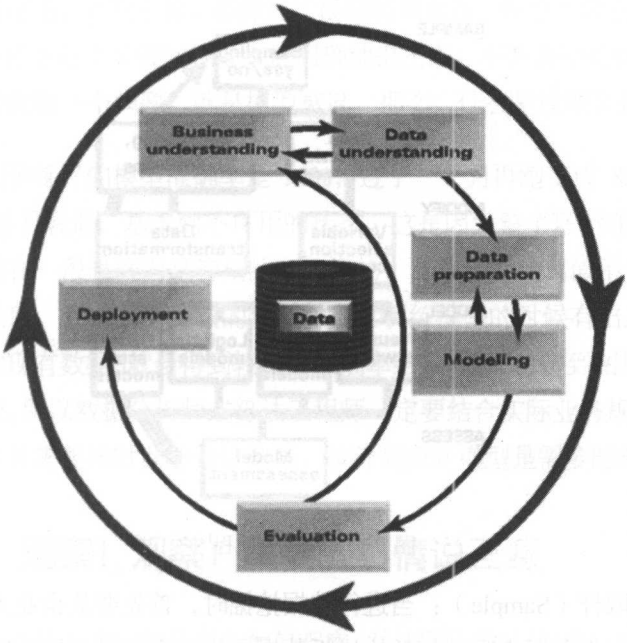


图 1-3 CRISP-DM 流程

对这个流程图中的每个步骤有如下解释。

- 商业理解（Business Understanding）：找问题从而确定商业目标。
- 数据理解（Data Understanding）：确定数据挖掘所需要的数据、数据的初步探索。
- 数据准备（Data Preparation）：选择数据、清理数据、调整数据格式使之适合建模。
- 建立模型（Modelling）：选择数据挖掘模型。
- 模型评估（Evaluation）：确定下一步怎么办，发布模型？还是对数据挖掘的过程进行进一步的调整，产生新的模型？
- 模型发布（Deployment）：把数据挖掘的结果送到相关人员手上、对模型

进行日常监测维护。

（2）SAS 公司的 SEMMA（如图 1-4 所示）。

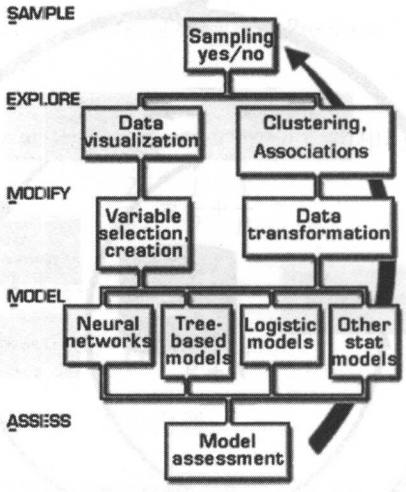


图 1-4　SEMMA 流程[1]

- 数据取样（Sample）：当进行数据挖掘时，首先要从企业大量数据中取出一个与你要探索问题相关的样板数据子集，而不是动用全部企业数据。通过数据取样，要把好数据的质量关。
- 数据特征探索、分析和预处理（Explore）：这里的数据探索，就是我们通常所进行的深入调查的过程。一开始，可以先观察众多因素之间的相关性；再按其相关的程度，以了解它们之间相互作用的情况。这些探索、分析，并没有一成不变的操作规律性；相反，是要有耐心地反复地试探，仔细地观察。
- 问题明确化、数据调整和技术选择（Modify）。
- 模型的研发、知识的发现（Model）。
- 模型和知识的综合解释和评价（Assess）：从上述步骤中会得出一系列的分析结果、模式或模型。同一个数据源可以利用多种数据分析方法和模型进

1　图片来源：https://www.packtpub.com/books/content/warming。

行分析，Assess 的目的之一就是从这些模型中自动找出一个最好的模型，另外就是对模型进行针对业务的解释和应用。

以上两种方法论，大同小异，都探究了模型前期准备、模型的建立以及模型的评测上线过程，但其中有个关键问题没有被明确地提出来：不管做分类也好，数据挖掘也好，最后都要面临一个问题，就是模型衰退。那么，什么是模型衰退呢？

比如我们刚训练好的模型准确率是 90%，过了一个月再跑变成 85%，再过一个月变成 80%，逐月衰退，基本到不可用的状态。这是因为整个环境在发生变化，训练好的模型可能在一段时间内是管用的、可靠的，准确率的偏离值不会超过 ±2%，但在环境变化以后，我们业务面临的数据和当时训练模型的时候有出入[1]，再用此模型去预测或分类现有数据，就会得到较低的准确率结果。后面会讲到图片识别的内容，也是同样的道理。所以数据分析师或算法工程师一定要结合实际业务规律给出模型更迭的计划表，并且需要按时调整该计划表。按计划更新模型是需要时刻关注的问题。

1.3 观察！观察！观察！重要的事情说三遍

对于英特来说最大的问题是学会如何观察，并让团队的其他研发工程师也具有这种能力。这种观察既观测了事物的外在特点，又考虑到事物的本质变化。学会这种观察并看到本质是人类最为独特的切入点，那一般的研发工程师能否依靠训练初步具备这种能力呢？很简单，多训练，多观察。

以一部大家熟知的美剧《越狱》为例来做些说明。片中有个情节是主角要去偷公司的数据，这个数据存在卡上，但卡被对方随身带着，怎么偷呢？韩国人比较厉害，做了个偷数据的工具，只要靠近卡就能读取数据。但如何找到这个人呢？警察马洪（Mahone）说我见过这个人的司机，他先对这名司机做了分析：总是标准站立姿势，手背后，以此推断他应该是位军人或曾经在部队服役；他总是穿着非常昂贵的西服套装，如果做一般工作的话是买不起的，所以只能是被 boss 长期雇佣的司机——由此推断只要找到该司机就能找到 boss。再看 boss 的车：品牌是劳斯莱斯，但比一般的

1 即使在线实时更新也不能避免模型衰退现象，只是能让模型衰退更晚到来。

劳斯莱斯底盘要低，因此有可能加装了防弹等设备，而美国能加装这些设备的公司一共也没有几家。再从司机的身高、体重全方位估算，就把这个人的基本画像做出来了，再去数据库里查找，就不再是大海捞针了。这说明观察或仔细观察在任何场合都是一项重要技能。

还记得在《神经网络与深度学习》这本书中提到过如何挑选橙子吗？下面我们将范围缩小些，就从如何挑选出口感更甜的橙子来做说明。

首先是能看到的信息。橙子的颜色和橙子的大小影响橙子甜度，接下去我们深入地想想，橙子软硬程度是不是也会有些影响？橙子的售卖时间决定了橙子的生长周期当然也有影响，进一步联想到橙子的产地。还有哪些影响？可能同一车运过来的橙子也有甜度的区分，那是否跟采摘的优先级有关，在树上的位置有关？

当然说来说去，橙子的品种也至关重要，出生论嘛。还有什么和橙子甜度相关的信息？气味？嗯，气味也是一种影响因素。

好了，有以上这些信息后，我们再看看如何获取这些信息？对于获取不到的信息能否推算出来。最后我们可能保留一些属性作为基本特征使用。

颜色、大小、软硬、发售时间、品种、产地，能直接得到的也就这些，对于这些属性，我们仔细地挑选一些橙子，并且标记出以上提到的颜色、大小、软硬等属性，并把它们做相关度评分。用 NumPy 中的协方差（COV）[1]和相关系数（CORRCOEF）来衡量相关程度。

$$\mathrm{COV} = \frac{\sum_i^N (Data_1[i] - Mean_1) * (Data_2[i] - Mean_2)}{N}$$

$$\mathrm{CORRCOEF} = \frac{COV}{STD_1 * STD_2}$$

协方差的绝对值越大表示相关程度越大，协方差为正值表示正相关，为负值表示负相关，为 0 表示不相关。这个函数非常简单，使用 NumPy 计算协方差和相关系数如下。

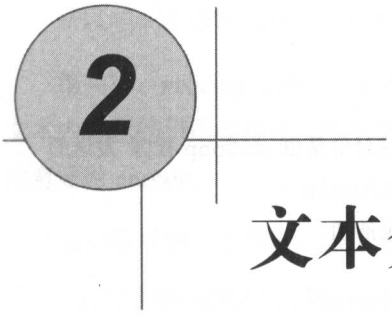
1 协方差只从统计学角度衡量相关度关系。无论何种统计方法，只是给出参考，勿以此为决定指标。

```
from numpy import array, cov, corrcoef

data = array([data1, data2...])
#计算两组数的相关系数
#返回结果为矩阵，第 i 行第 j 列的数据表示第 i 组数与第 j 组数的相关系数。对角线为 1
corrcoef(data)
```

我们能得到一个关系矩阵，这个矩阵能告诉我们哪个系数与最终的值（这里是橙子的甜度）相关。这样我们从中找到相应的参数，先用这些有正相关或负相关的参数作为特征，用作下一步模型的输入数据。

不同维度的观察能带来不同的思考，我们把人看问题的角度变成机器看问题的角度，这就是机器学习的本意。



文本分析实战

2.1 第一个文本问题

英特开始复盘第一个文本分析问题：邮件分类问题。

对于邮件分类问题，我们还是把该问题抽象为一个文本分类问题，先不看其他影响垃圾邮件判断的因素，比如邮件发送的时间，同一 IP 发送的邮件，邮件发送人等等。现在我们将思维只集中在我们的文本上，对于邮件来说，我们有两个类型的文本可以提取：主题（标题）和邮件内容。

2.1.1 邮件标题的预处理

算法团队开始检查第一个关键点，邮件标题。现在的目标简化为如何通过邮件标题区分垃圾邮件和正常邮件。先来看看垃圾邮件标题的特征，如下所示。

(AD)蒲公英新品 X5 首发：你的第一台"跳蛋"路由器

016 IBM 云计算峰会报名免费抢座-2016.10.19 北京国际饭店会议中心

Can We Talk? Setting the Record Straight About 4 Content Misconceptions

欢迎参加简仪科技开源测控技术研讨会-北京站 2016 年 9 月 22 日

```
NEW! Advance your career with 19 MicroMasters programs
10.1 元购家电！iPhone6s-Plus 仅 4488，海尔热水器 699~10 亿优惠券狂撒，国庆福利快接着→
```

再看看正常邮件标题长什么样，如下所示。

```
[WeLoop 社区] 论坛注册地址 - 论坛注册地址 这封信是由 WeLoop 社区 发送的
Fwd: [Bitbucket] SSH key added to futurenlp
您的帐户在 Linux 设备上的 Chrome 中有新的登录活动
神经网络与深度学习代码
Fw:关于 ICP 备案申请审核通过的通知
技术部-SSL 数字加密证书
```

从以上例子可以看出垃圾邮件的标题特征有它的特殊性，但不能简单地通过是否包含什么关键词来判断是否为垃圾邮件。比如“云计算”出现在垃圾邮件中，但这个词很有可能也存在于正常邮件中。

不管用什么算法，文本处理的第一步都是提取特征。提取特征的前提是从各个角度观测要处理的事物，它们可能是一段文本，也可能是一幅图像。上面的示例是一段文本。而对于大部分的文本处理来说，都可以将词作为文本的最小粒度。只是单词还无法完全说明这段文字的含义，如果非要切换成词来看的话，这个词的前后文字也非常重要，当然重要的还有字符出现的频率、英文、符号等。

饭要一口一口地吃，先不要纠结这些特征，集中到最小粒度集合上，这就是读者常常听说的“分词”了。对于人类来说，在说话或读书时，天然有断句、断词的本领，但计算机到今天为止，还没有很好地学会这项本领。由于本书主要讲深度学习应用，不涉及分词的具体原理和技术细节，在此略过。下面我们直接选用 Jieba 分词包来做分词。

```
#coding=utf8
import jieba
title_words = jieba.lcut(u'10.1 元购家电！iPhone6s-Plus 仅 4488，海尔热水器 699~10 亿优惠券狂撒，国庆福利快接着→')
for x in title_words:
print x.encode('utf8')
```

程序虽然简单，但在 Python2.7 里面总是会有字符转码的问题。

所以咱们花点时间来说说如何操作 Unicode。

第一行，#coding=utf8 或 # -*- coding:utf-8 -*- 是文档的编码标识，这样在程序文本内显示的引用或操作一些中文字符就不会出现错误。最后一行，在 print 输出时注意要做 encode。

分词后得到一个列表，为了节省篇幅，我们把列表横向展现出来。

```
10.1/元购/家电/！/iPhone6s/-/Plus/仅/4488/，/海尔/热水器/699/~/10/亿/优惠券/狂撒/，/国庆/福利/快/接着/→/
```

英特很快从中发现了以下问题。

（1）标点较多。

（2）有一些词出现频率太高，就被认为是无意义的。

下面针对以上问题分别进行处理。

（1）标点：这里我们将‘，’‘！’‘：’‘。’等标点都删除，是因为在这个任务中，我们认为标点符号没有字的特征有用。在后文情感分类中，有些标点就不能在预处理时删掉，因为我们认为这些标点是有用的，比如‘！’‘？’等。所以什么时候去掉标点，去掉哪些标点，都要根据你的任务来决定，总的原则是尽可能多地去除噪音，尽可能多地把决定性特征保留下来。

（2）对于以一些诸如‘仅’、‘元’、或者其他的一些‘我’、‘这’、‘那’这类的词，它们经过人的粗略判断认为对于垃圾邮件的识别没有帮助，都属于在下一步的提取特征时用不到的内容，可能还会起到噪音影响，因此可以连同标点一起删除。在此我们采用停词表的方式来删除。

> **提醒：**在做文本处理时必须维护好两张表，第一张是你需要删除的停用词表，第二张是考虑到任何分词工具都有缺陷，需要准备的一张新词表，便于及时补充新词而适应业务发展，而对于大多数业务而言，在针对不同领域时，还必须维护针对不同领域的新词表。

回到停用词表，它可以用一个文本文件表示，比如 stopwords.txt 这样的文件名，基于这个停词表，我们使用 not in 将停词删除。

```
    words = pseg.lcut(content)
    for word, flag in words:
        # print word.encode('utf-8')
        if (word not in customstopwords):    #去停用词
result += word.encode('utf-8')
```

算法组在 Windows 系统下也遇到了问题，当你将邮件的标题当做文件名写入时，就会报非法字符错误。

所以，你需要去除一些非法字符。

```
# 去除标题中的非法字符 (Windows)
def validateTitle(title):
    rstr = r"[\/\\\:\*\?\"\<\>\|]"  # '/\:*?"<>|'
    import re
    new_title = re.sub(rstr, "", title)
return new_title
```

有时也会需要去除制表符等。

```
#一次性去除空格,换行符,制表符
"".join(s.split())
```

最后，我们得到一个这样的字串（根据你的停词不同可能结果不同）。

```
10.1 购家电 iPhone6sPlus4488 海尔热水器 699 亿优惠券狂撒国庆福利快
```

这里的字串将作为下一步的输入，请继续往下看。

2.1.2 选用算法

回到第一步的算法思维，请读者思考下一步我们应该做些什么？如果你还不确定你的思路，可以看看英特是怎么做的，英特现在已去除了停用词和一些不常用词，那么接下来就到特征抽取及选取模型的阶段了，整个文本分类的训练模型最简流程如图 2-1 所示。

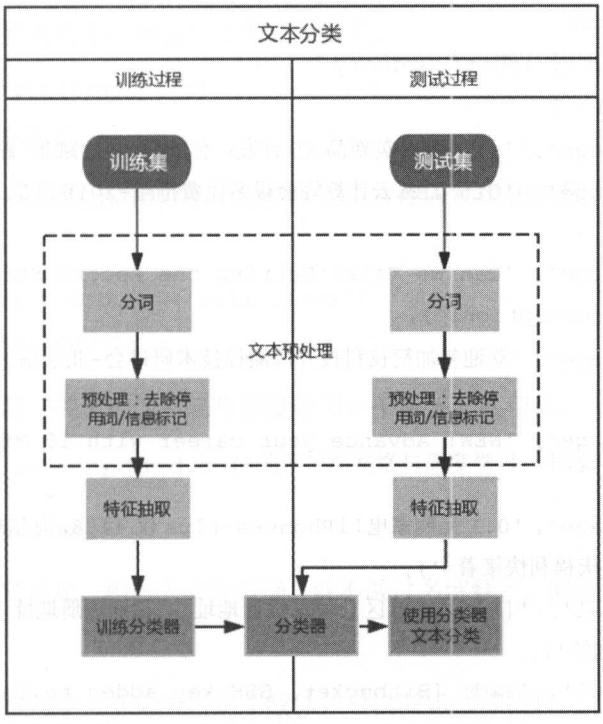


图 2-1 文本分类流程图

对于文本的特征抽取及表示，我们有很多方法，本小节暂不介绍，我们先把整个算法的流程走完，现在请将目光集中在所使用的分类器上。

常用的分类算法，想必大家很熟悉的有贝叶斯、SVM、决策树等，选用哪种需结合实际情况来定，我们这里使用一种多分类算法（排除决策树）：SVM 在文本分类上是一个比较好的选择。

如果觉得直接使用 sklearn 的 SVM 或 SVC 有点复杂，这里给出一个开源的 SVM 文本分类封装包 TextGrocery，该封装包集成了对文本的一些处理，适合初学者，地址：https://github.com/2shou/TextGrocery。

下面，我们将分类输出定义成['garbage', 'normal']，并将之前的邮件语料样本放进去试试，来看一段代码。

```
#coding=utf8
from tgrocery import Grocery
train_src = [
    ('garbage', '(AD)蒲公英新品 X5 首发：你的第一台"跳蛋"路由器'),
    ('garbage', '016 IBM 云计算峰会报名免费抢座-2016.10.19 北京国际饭店会议中心'),
    ('garbage', 'Can We Talk? Setting the Record Straight About 4 Content Misconceptions'),
    ('garbage','欢迎参加简仪科技开源测控技术研讨会-北京站 2016 年 9 月 22 日'),
    ('garbage', 'NEW! Advance your career with 19 MicroMasters programs'),
    ('garbage','0.1 元购家电！iPhone6s-Plus 仅 4488，海尔热水器 699~10 亿优惠券狂撒，国庆福利快接着→'),
    ('normal', '[WeLoop 社区] 论坛注册地址 - 论坛注册地址 这封信是由 WeLoop 社区 发送的'),
    ('normal', 'Fwd: [Bitbucket] SSH key added to futurenlp'),
    ('normal', '您的帐户在 Linux 设备上的 Chrome 中有新的登录活动'),
    ('normal', '神经网络与深度学习代码'),
    ('normal', 'Fw:关于 ICP 备案申请审核通过的通知'),
    ('normal', '技术部-SSL 数字加密证书')
]
# 创建一个 grocery，'mail_class'为模型名称
grocery = Grocery('mail_class')
grocery.train(train_src)
grocery.save()
# Load model(和之前设的名字一样)
new_grocery = Grocery('mail_class')
new_grocery.load()
# 预测
print new_grocery.predict('关于神经网络与深度学习一书源码')
```

我们在准备好数据后，就可以用 TextGrocery 处理了，TextGrocery 集成了提取特

征算法，所以直接调用并存储就可以生成模型了。

```
grocery.train(train_src)
grocery.save()
```

然后我们将模型取出并预测，这里为了区分存入和取出的模型用了新的变量new_grocery。

```
new_grocery = Grocery('mail_class')
new_grocery.load()
```

我们尝试预测“关于神经网络与深度学习一书源码”这句话。

```
print new_grocery.predict('关于神经网络与深度学习一书源码')
```

预测结果为 normal。

这段代码非常简单，相信大家能看懂，就不做过多解释了。但这里要注意以下三点。

（1）训练语料太少，准确率肯定比较低，可以尝试增加语料。

（2）相信细心的读者已经看出来了，这段代码是没有做文本预处理的。

（3）在这个过程中，使用其他算法或提取特征的方法，是否会增加准确率？

这三点英特及团队后期经过改进全部解决了，第三点就留给大家当一个小练习吧，看能把准确率提高到多少？期待你们的表现。

2.1.3 用 CNN 做文本分类

以上我们故意忽略了文本特征提取部分，因为对于一般读者而言，能使用工具或库就可以了，但相信一些对自己有更高要求的读者想知道怎么提取文本特征。所以这里稍微提一下，TextGrocery 是怎么做特征提取的，以及提取的都是哪些特征。

我们一起来看看源码部分：先分词，再建立一个字典数据类型（词对应着词的序号），然后以 Unigram 和 Bigram 的方式分别提取特征（这两种方式后文会提到），最后将这些特征送入 SVM 分类器。

以下节选代码展示了如何做预处理。

```
def preprocess(self, text, custom_tokenize):
    if custom_tokenize is not None:
        tokens = custom_tokenize(text)
    else:
        tokens = self._default_tokenize(text)
    ret = []
    for idx, tok in enumerate(tokens):
        if tok not in self.tok2idx:
            self.tok2idx[tok] = len(self.tok2idx
        ret.append(self.tok2idx[tok])
    return ret
```

以下代码展示了如何用 Unigram 和 Bigram 提取特征，这两个方法在类 GroceryFeatureGenerator 中。

```
def unigram(self, tokens):
    feat = defaultdict(int)
    NG = self.ngram2fidx
    for x in tokens:
        if (x,) not in NG:
            NG[x,] = len(NG)
        feat[NG[x,]] += 1
    return feat

def bigram(self, tokens):
    feat = self.unigram(tokens)
    NG = self.ngram2fidx
    for x, y in zip(tokens[:-1], tokens[1:]):
        if (x, y) not in NG:
            NG[x, y] = len(NG)
        feat[NG[x, y]] += 1
    return feat
```

好了，在以上案例中我们了解了 TextGrocery 怎么做特征提取，特征提取之后使用了一个 SVM 分类器做分类。那么深度学习中是否有很好的方法，同时将特征提取和分类器都替代掉呢？

这里介绍一种深度学习的用法——CNN 的文本特征提取，并最后输出一个一维分类层，完成了从特征提取到分类的整体模型。为简化难度，我们可以使用 https://github.com/dennybritz/cnn-text-classification-tf.git 开源库。同上，我们仍然选取两类特征来区分垃圾邮件和非垃圾邮件，步骤如下。

（1）选取最长的句子当作向量长度，比如最长的句子有 350 字符，接着把所有的字符按其在字典中的序号填入，句子长度不够 350 时其他位补 0，生成如下的向量表示。

```
[[ 1  2  3 ...,  0  0  0]
 [ 1  3 10 ...,  0  0  0]
 [ 1  2  3 ...,  0  0  0]
 ...,
 [ 3  9  3 ...,  0  0  0]
 [ 9  2  2 ...,  0  0  0]
 [11  2 11 ...,  0  0  0]]
(19801, 350)
Vocabulary Size: 588
Train/Dev split: 17821/1980
```

（2）将训练集和测试集分开并开始训练。

```
2016-12-08T20:02:08.710200: step 1, loss 2.24919, acc 0.484375
2016-12-08T20:02:10.192376: step 2, loss 2.35938, acc 0.46875
2016-12-08T20:02:11.664114: step 3, loss 2.44823, acc 0.484375
2016-12-08T20:02:13.860079: step 4, loss 2.52388, acc 0.453125
2016-12-08T20:02:15.522624: step 5, loss 1.9896, acc 0.5
2016-12-08T20:02:17.005463: step 6, loss 2.12384, acc 0.515625
2016-12-08T20:02:18.434234: step 7, loss 1.61683, acc 0.59375
```

训练的起始阶段 acc 比较低，从 0.484 开始 7 次之后已经有 0.59375，而 loss 下

降到 1.6。

继续到 589 次我们得到一个相对比较好的成绩。

```
2016-12-08T20:18:00.183656: step 589, loss 0.197619, acc 0.9375
```

训练完成后，用得到的模型可以直接做句子的预测。这里就不再赘述，感兴趣的读者可以自己动手试试。

那么，CNN 是怎样提取文本特征继而做分类的呢？我们会在 2.3 小节中给出相关解释。

2.2 情感分类

针对邮件分类的问题，英特团队在发现症结后，处理得很好。业务部门这次交给英特一个真正产品上遇到的问题：区分用户发帖的情感。根据用户的情感和用户所在的分论坛，就能大概地确定有多少用户喜欢这个论坛里讨论的东西，有多少用户不喜欢。比如“汽车之家”论坛，用户在奥迪 A4 这个论坛里面发表了“发动机有问题，总是烧机油”等负面评价，如果识别为负面信息，那对于奥迪 A4 而言，它在网络上的负面信息又多了一条。通过这样的分析，能区分对于某个特定车型，用户是喜欢还是厌恶。这对很多行业来说都是比较基础的通用需求。

2.2.1 先分析需求

英特看了下需求，发现用户的情感是针对产品的，目前只需要对产品或产品的某一个部件，某一类品质进行评价，这里的情感倾向只需要三类，支持、反对、中立，广义上也可以说是正面、负面、无情感（中立）。对于另外一些应用来说，情感倾向的强弱程度也是比较重要的。

情感倾向分析的方法主要分为两类：一种是基于情感词典的方法；一种是基于机器学习的方法。前者需要用到标注好的情感词典，英文的词典有很多，中文的话，主要有知网整理的 Hownet 和台湾大学整理发布的 NTUSD 这两个情感词典，另外，哈工大信息检索研究室开源的《同义词词林》也可以用于情感词典的扩充。基于机器学习的方法则需要大量人工标注的语料作为训练集，通过提取文本特征，构建分类器来

实现情感的分类。

这两者各有什么优劣？英特团队也进行了一番调查。

（1）基于情感词典的方法，优点是非常稳定，如果一句话中包含这个词就提取出来，然后做正负向情感分类，只要有就能鉴别出来。而这种方法的缺陷在于，对于一些不在情感词表中的单词或表示方法就完全无能为力。

比如“这个发动机烧机油啊！”这句话没有包含常见的情感倾向词，但我们一看就知道他说的是负面消息，因为人们已经建立了这种知识体系，知道了烧机油=说汽车的问题=情感负向，进而判断出情感倾向性。由于只根据情感词判断，忽视了语料语境的作用，同一个词在不同的语境中表达的意思完全不同，有时候甚至会影响情感表达。

（2）机器学习是如何做的呢？机器学习需要人工标注语料作为训练集，提取出文本的特征，用特征构建一个分类器，再做情感的分类。因为在构建特征分类器时加入的文本特征包含一部分语境，部分规避了情感词典的弱点。但这种方式因为特征的抽取方法不同，而导致有噪音，通用性不好。假设用汽车论坛语料训练出的模型去预测某个手机商品评论，就会出现错误率很高，根本不可用的情况。

为了弥补这两种方法的缺陷，现在业界通用的有三种处理方法。

（1）利用已有知识结构自动学习各种语料。这种处理方法的简单应用有很多，比如利用维基百科训练一个 word2vec 模型，每个词的向量实际关联了很多意义，再使用词向量表示句子，将会规避一部分问题。而知识图谱的建立、语境的感知在情感分析中都是非常必要的。

（2）结合词法分析和机器学习两种方法做综合判断，以减少误差。

（3）汽车领域的模型无法用在其他领域，某种程度上是由于训练集不够引起的，最好的处理办法是每个领域，每个行业都用行业的语料重新训练一个模型，并定期更新模型。如果做不到，就尽可能地增加语料范围。

2.2.2 词法分析

英特首先选择了基于情感词典的方法进行实验。

因为评论里可能包含好几句话，情感的计算必须要选择一个最小单位，我们就按照“，”为切分单位，将一句话或一段话按需切分。

```
punc = ["。","; ","? ","! "]
Sentence = content.split(punc)
```

按“，”切分成句子的最小单元。

```
Group = Sentence.split(",")
```

接下来，我们基于已有的中文情感词库，构建一张情感词表，然后对文本进行中文分词处理，将处理后得到的单词依次在预先构建好的情感词表中逐个配比查找，如果出现在表中，则是情感词，并读取情感极性及相应权值，否则就不是情感词，并进入下一个候选单词，直至整句话判断结束。

```
If word in senDict:
    senWord = (句中位置，情感倾向，情感强度)
```

由于我们有四张词表，所以稍微复杂点，这四张词表为别是正向词表（positive），负向词表（negative），程度前缀词表（very_prefix）、否定词表（negation）。

正向词表和负向词表因为没有分数，所以我们给一个默认值，负向情感的是-0.08，正向情感的是+0.08。

正负向词表、前缀词表里有相应的分数，直接取分数即可，否定词表则取反。

```
for currWord in set_negative_word:
if currWord not in dict_word_score.keys() and
currWord.endswith(u'的') == False and currWord != u'东西':
        dict_word_score[currWord] = -0.08
for currWord in set_positive_word:
    if currWord not in dict_word_score.keys() and
currWord.endswith(u'的') == False:
        dict_word_score[currWord] = 0.08
```

由于汉语中存在多重否定现象，而一旦出现否定词后，和否定词关联的情感词表达的情感极性也会发生反转，比如“你穿的花裙子真好看啊！”“你穿的花裙子真不

好看啊”，如果将“好看”做为一个情感极性词，我们需要找到修饰它的副词中是否包含否定词，第二句话中的“真不好看啊”就是在情感词前面加入了否定词，导致整句话变成负向情感。对于多重否定我们要做的是在否定词前面再去发现否定词，如果又找到否定词，这就是双重否定，整句话的意思就表达了原有的情感词的情感极性；双重否定还有种情况，类似于出现“不得不”这样的双重否定词。下面我们来看一段专门处理否定问题的伪码。

```
# 需要准备的词典：
#   正向词典：    pos_dict
#   负向词典：    neg_dict
#   否定前缀：    neg_prefix_dict（如：不是,不算 等）
#   双重否定前缀：double_neg_prefix_dict （如：不能不,不得不 等）
#   程度前缀：    very_prefix（如：最,很 等）

# curr_index 表示当前情感词在句子中的位置
def prefix_process(curr_index, sentence, score):
seg = sentence[curr_index-5: curr_index]
# 双重否定
    for curr_neg_prefix in double_neg_prefix_dict:
    if seg.endswith(curr_neg_prefix):
        return 0.8 * score

# 否定前缀直接修饰情感词
    for curr_neg_prefix in neg_prefix_dict:
        if seg.endswith(curr_neg_prefix):
            temp_pair = pseg.lcut(sentence[0:curr_index])
            # 判断是否有双重否定修饰
            for i, (w, f) in enumerate(reversed(temp_pair)):
                if 标点符号:
                    break
                elif 代词 / 名词 / 数词（找到情感词修饰的主语）:
                    if 情感词的主语被否定（因为修饰否定的否定词已经处理过
了）:
```

```
                return 1.3 * score
            return -1.3 * score

        # 否定前缀修饰情感的主语
        for i, (w, f) in enumerate(reversed(temp_pair)):
            if 标点符号:
                break
            elif 代词 / 名词 / 数词（找到情感词修饰的主语）:
                if 情感词的主语被否定（因为修饰否定的否定词已经处理过了）:
                    return -0.6 * score

        # 程度前缀
        for curr_very_prefix in set_very_prefix:
            if seg.endswith(curr_very_prefix):
                return 1.3 * score

    return score
```

词法分析的情感分析因为其规则性非常强，甚至能处理类似“我发现我的死对头被男朋友甩了”这种隐含的多重否定句。当然这里的规则与关联规则比较多，读者可以尝试更好的解决方案。

2.2.3 机器学习

机器学习的情感分类的分类器可以使用第一章我们谈到的 SVM 分类器 TextGrocery，而对于机器学习的特征提取，则需要更仔细些。

比如说，一句话中我们只取形容词、副词、名词、否定词等，而不取其他的词性，因为其他词性对于情感分类来说没有帮助，或者是非主要特征。

以下代码显示了如何使用 TextGrocery 提取训练集并训练模型的过程。最后将这个模型存储到磁盘上的"./algorithm/saved_model/svm_sentiment.model"路径下并返回。

下面这段代码演示了如何用 TextGrocery 训练一个模型。

```
def sentiment_train(train_set):
```

```
        """
        训练模型
        :param train_set: 训练集
        """
        gro_ins =
Grocery("./algorithm/saved_model/svm_sentiment.model")
        gro_ins.train(train_set)
        gro_ins.save()
        return gro_ins

    def get_xlsx_labeled_data(path, col_tag=0, col_content=1):
        """
        读取训练集
        :param path: 训练集路径
        :param col_tag: 标签的位置
        :param col_content: 内容的位置
        :return: 列表,[[tag, content],[tag, content]]
        """
        results = pd.read_excel(path, header=None)
        contents = list(results[col_content])
        train_data = []
        for i in range(len(contents)):
            result =
strip_tags(clean_comment(delete_stop_words(contents[i])))
            train_data.append([results[col_tag][i], result])
        return train_data
```

过滤词性。

```
        # 需要过滤的词性
        if is_filter:
            filter_polar = [u'n', u'f', u'a', u'z']
```

加载数据。

```
# 加载数据
train_data = get_xlsx_labeled_data('./data/car/train.xlsx')
```

最后我们调用训练函数进行训练。

```
# 训练模型
model = sentiment_train(train_data)
```

这段代码和邮件分类差不多，就不做过多解释了。

2.2.4 试试 LSTM 模型

针对文本的特征提取，一直到今天都还有层出不穷的各种方法讨论。在深度学习中利用 RNN 也能提取文本特征。下面我们给出一段 LSTM 提取特征的实例，看看如何用 LSTM 提取特征并训练模型。

```
# -*- coding: utf-8 -*-
"""
lstm 模型
"""
from keras.utils import np_utils
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.layers.embeddings import Embedding
from keras.layers.recurrent import LSTM


def lstm_train(dic, x, y, maxlen):
    print('Build model...')
    model = Sequential()
    model.add(Embedding(input_dim=len(dic) + 1, output_dim=256,
input_length=maxlen))
    model.add(LSTM(128))  # try using a GRU instead, for fun
    model.add(Dropout(0.5))
    model.add(Dense(3))
    model.add(Activation('sigmoid'))
```

```
        model.compile(loss='binary_crossentropy', optimizer='adam',
class_mode="binary")
        model.fit(x, y, batch_size=32, nb_epoch=10,
show_accuracy=True)  # 训练时间为若干个小时
        return model
```

目前在应用时如果没有特别需求，一般用的是 BiLSTM ，也就是双向的 LSTM。双向 LSTM 因同时做了正向句序和反向句序的特征抽取，为后期处理提供了更多的可用素材，保留了更完整的句子特征。

那么 LSTM 或 BiLSTM 与 CNN 比较，哪种提取特征的方式更好呢？有一篇论文[1]专门论述了各种 RNN 和 CNN 的比较，得出的结论是各有各的长处，有兴趣的读者可以读一读。

2.3 文本深度特征提取

完成两个项目后，英特深深地感觉到无论做文本分类还是预测，文本特征的提取都非常重要。而文本特征的提取说白了就是如何将自然语言理解的问题转化成机器学习的问题。第一步肯定是要找一种方法把语言表达数学化，即用可量化的方式来表示文本的特征。下面我们跟随英特一起来看看文本的深度特征是如何量化表达的。

2.3.1 词特征表示

文本的深度特征有四种表示方法，下面我们逐一讲述

1. 词表法（字典法）

先来看第一个表示方法——词表法。词表法是指将需要处理的多个文档中所有的词进行剔重、排序，形成一个词表也叫字典，用这个词表中对应的序号来表示一个句子或文档中出现的词。

1 *Comparative Study of CNN and RNN for Natural Language Processing*, Wenpeng Yin, Katharina Kann , Mo Yu and Hinrich Schutze 2017.2.7。

例如“你今天吃过饭了吗？”这个句子中出现的词能在词表中对应索引，后续则使用这个索引在词嵌入矩阵中提取对应的词向量。

```
('你', 1)
('今天', 7)
('吃过', 34)
('饭', 5)
('了', 2)
('吗', 4)
('？', 189)
```

“你今天吃过饭了吗？”这句话的向量表达就是[1,7,34,5,2,4,189]，这样就方便扔到模型中做计算了。

一般来说，对于 Embedding 层的输入基本上都使用词表法处理后的向量表达，这也是为什么我将这个方法放在第一个来谈的原因。Embedding 层是什么？以下是本书作者在知乎上的一个回答。

Embedding 层是什么？

Embedding 层可以参考 word2vec 或 glov 算法原理，利用单层神经网络做词的向量化，一般来说输入为词在字典中的位置（一般不用 One-Hot），输出为向量空间上的值。

在 Embedding 这种结构出现之前，一般先用 word2vec 计算词向量，然后将词向量作为模型的输入层，计算词向量部分和模型是两个部分，而 Embedding 出现后就将这两个部分合并在一个模型中，输入层数据不是词向量，而是词在字典中的位置。Embedding 主要不是作为降维使用[1]，而是作为一种特征表示使用。

---

2. One-Hot 表示

接着看下一种词表示方式 One-Hot Representation，它是最直观的词表示方法之一。

1 当然 Embedding 有降维的作用。

这种方法把每个词表示为一个向量，这个向量的长度就是文档（句子）的长度。在向量中该词出现的位置填 1，其他位置都填 0，那么这个向量就代表了当前的词。

举个例子，在“明天你会买这本书吗？”这个句子中，用本方法则可以把“明天”表示为 [0 0 0 1 0 0 0 0 ...]，“你”表示为 [0 0 0 0 0 0 0 1 0 ...]。

但这种存储方式明显带来了存储空间浪费很大，矩阵明显稀疏的问题，这不利于存储和处理，也就是我们常说的“维度灾难”。于是，我们换一种表示方式，设“明天”的序号为 1，“你”的序号为 2，“会”的序号为 3……依此类推。这就是我们在上一节中提到的词表法了，这样虽然解决了稀疏矩阵的问题，但又产生了新问题。我们在说一句话的时候，词-词之间的关系在某种意义上是否可以替换？比如说“明天”，那它是否和“今天”有关联？在 One-Hot 的表示方法中是没有词-词之间关系的。这样，我们就需要一种新的表示方法，不但能标识这个词，也能标识出词与词的关系。而词与词的关系衡量，我们可以转换成衡量词与词的距离。

3. n-gram 模型

还记得使用 TextGrocery 处理邮件时用到的特征提取 Unigram、Bigram 吗?这里我们要讲的正是这种特征提取方法。

n-gram 模型也叫 N 元模型，是自然语言处理（NLP）中一个重要的概念，它突出的是一个组的思想，1gram 可以是一个单词或一个中文汉字，比如假设有一个字符串 s，那么该字符串的 n-gram 就表示按长度 N 切分原词得到的词段，也就是 s 中所有长度为 N 的子字符串。通常在 NLP 中，我们基于一定的语料库，可以利用 n-gram 来预计或者表示一个句子，另外一方面 n-gram 也用来评估两个字符串之间的差异程度。

还记得之前看到的 TextGrocery 源码吗？Unigram、Bigram 就是属于 n-gram 的子集，这里的 N 可以是任意数量。Unigram 指的是单字，而 Bigram 指的是双字。

下面我们一起来看看 n-gram 算法是如何实现的，以下代码在 NLTK 库中。

```
def ngrams(sequence, n, pad_left=False, pad_right=False,
        left_pad_symbol=None, right_pad_symbol=None):
```

```
    sequence = pad_sequence(sequence, n, pad_left, pad_right,
                            left_pad_symbol, right_pad_symbol)

    history = []
    while n > 1:
        history.append(next(sequence))
        n -= 1
    for item in sequence:
        history.append(item)
        yield tuple(history)
        del history[0]
```

sequence 依据 n 的大小输出 gram，将 sequence 以一个元素加入到 history 列表中，再返回该列表。

举个直观的例子来展示下 n-gram 如何实现：输入为“鲁迅老师生前是一位革命家，死后仍然是。”将分词后句子输出。

句子分词结果如下。

鲁迅/老师/生前/是/一位/革命家/，/死后/仍然/是/。

调用 ngram 函数，定义 n=3 时的输出结果如下。

鲁迅/老师/生前
老师/生前/是
生前/是/一位
是/一位/革命家
一位/革命家/，
革命家/，/死后
，/死后/仍然
死后/仍然/是
仍然/是/。

在 n=3 时，我们叫它 trigram，在 n=2 时就叫 bigram，n-gram 事实是将句子的信息扩充了，n 越大扩充的信息越多。

这里需要提一下应用场景也很多的 skip-gram，无论它们之间是否有间隔词，“白色汽车”和“白色的汽车”会被识别为相同的短语，以 skip-gram 的特性，对低频词更加敏感。来看看 skip-gram 例子：首先，“skip”就是指需要跳过多少个字符，其参数也多了一个 k 来决定输入值需要跳过几个词（gram）。

skip grams, n=2, k=2，输出结果如下。

鲁迅/老师
鲁迅/生前
鲁迅/是
老师/生前
老师/是
老师/一位
生前/是
生前/一位
生前/革命家
是/一位
是/革命家
是/，
一位/革命家
一位/，
一位/死后
革命家/，
革命家/死后
革命家/仍然
，/死后
，/仍然
，/是
死后/仍然
死后/是
死后/。
仍然/是
仍然/。
是/。

我们学会用 n-gram 来表示句子后，第一个想到 n-gram 模型能解决的问题就是衡量两个相像的句子差异有多少？这种衡量要靠字符串距离确定，这就有必要了解“模糊匹配”的概念。

例如，如果我们要评估一篇文章中的关键词“神经网络”出现的次数，这时所使用的方法是精确匹配模式，与之相对的就是模糊匹配。模糊匹配的应用也随处可见。例如搜索引擎，如果你输入一个错误的查询单词后，比如将“四大皆空”输成“四大及空”，搜索引擎输入框会立即提示你是否要输入的词是“四大皆空”，甚至在输入法这个层面就会提示阻止你继续错下去。能完成这种纠正，肯定是先要将错误单词和正确单词进行映射，这种技术我们称为文本的模糊匹配。

模糊匹配的关键在于如何衡量两个相似单词的差异，这个差异用书面语言称为“距离”。这种基于距离的衡量有多种方法，而 n-gram 距离也是其中的一种。

有学者提出下面的公式来表示这种距离。

假设我们要比较字符串 s 和字符串 t，要算出这两句话的距离，可以应用下面的公式。

```
|s 中的 N-Gram 数| + |t 中的 N-Gram 数| - N* |s 和 t 共有的 N-Gram 数|
```

以上公式的数学表达如下。

```
|GN(s)|+|GN(t)|-2*|GN(s)∩GN(t)|
```

其中 N 值就是我们 n-gram 中 N，一般取 2 或者 3，这里我们以 N=3 为例对以下字符串进行分段，并求得其距离。

字符串 s=“鲁迅/老师/生前/是/一位/革命家/，/死后/仍然/是/”
字符串 t=“鲁迅/老师/生前/是/一位/革命家/和/作家/，有/很多/伟大/的/作品/”

字符 s 的 3-gram 就是 9，字符串 t 的 3-gram 就是 11，s 和 t 共有的 gram 为 4。“鲁迅/老师/生前/是/一位/革命家/”这一部分是完全一致，公共 gram 数量为 4，带入公式得（9+11）-3×4=8，则这两句话的距离为 8。

本节我们了解如何用 n-gram 来衡量词与词之间的距离，进而能衡量词与词的关系，除了用 n-gram 直接提取特征外，还有什么方法能更进一步，用一组固定向量来

表示一个词呢？词嵌入（Word Embedding）给了我们答案，我们在下一节“分布表示和词嵌入”中详细阐述。

4. 分布表示和词嵌入特征表示

Distributed Representation（分布表示，下同）最早由 Hinton 在 1986 年提出。它是一种低维实数向量，这种向量一般长成下面这种样子。

[0.792, −0.177, −0.107, 0.109, −0.542, …]

维度以 50 维和 100 维比较常见，当然了，这种向量的表示不是唯一的。

一段文本的语义分散在一个低维空间的不同维度上，相当于将不同的文本分散到空间中不用的区域。分布表示是文本的一种表示形式，具体为稠密、低维、连续的向量。向量的每一维都表示文本的某种潜在的语法或语义特征。

如何理解呢？就好像我有一个平面，所有的词分散在整个平面中，词的坐标是已知的，那么依靠在平面中的位置可以得到词-词的关系。

2003 年 Bengio 提出 NPLM 的时候，在模型中去学习每个词的一个连续向量表示，并经过 Tomas Mikolov 等人的发展，发展出 Word Embedding（词嵌入，下同。有的地方直接翻译成词向量，笔者认为词向量是一类概念的总称，遂没有采用）这一表示方法。上一节中那句话，用 Word Embedding 表示后，可能是下面这样的。

第一个词：[0.2 0.3 0.5]

第二个词：[0.7 0.1 0.2]

第三个词：[0.1 0.2 0.3]

第四个词：[0.2 0.3 0.4]

第五个词：[0.3 0.4 0.6]

……

Word Embedding 使寻找相关或者相似的词成为可能。向量的距离可以用最传统

的欧氏距离[1]来衡量，也可以用 cos 夹角来衡量。用这种方式表示的向量，“明天”和“今天”的距离会远远小于“明天”和“你”的距离，甚至在可能的理想情况下“明天”和“今天”的表示应该是基本一样的。

这样两个词之间就可以进行比较了。当然，在这个方法下相似度高的两个词，并不一定具有相同语义，它只能反映出两者经常在相近的上下文环境中出现。

前文提到在 Embedding 这种结构出现之前，一般先用 word2vec 计算词向量，word2vec 也是词嵌入的一种实现方式。它目前使用的神经网络模型有两种：CBOW 和 skip-gram，这两种模型都很简单粗暴（如图 2-2 所示）。

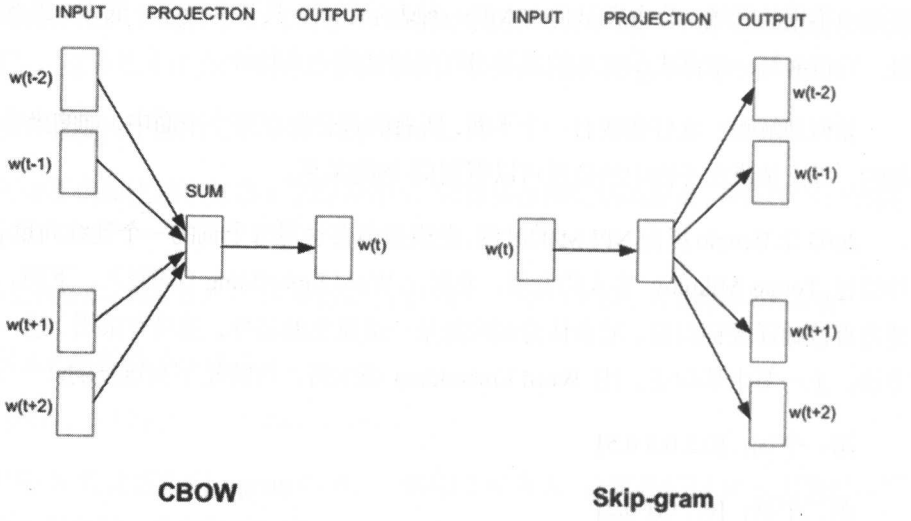


图 2-2　Word Embedding 的两种模型实现：CBOW（continuous bag-of-words）和 skip-gram

在 cbow 方法里，训练目标是给定一个 word 的 context，预测 word 的概率；在 skip-gram 方法里，训练目标则是给定一个 word，预测 word 的 context 的概率。skip-gram 的输入是当前词的词向量，而输出是周围词的词向量。也就是说，通过当前词来预测周围的词，而 cbow 模型正好是将输入输出调转。

1　又称为欧几里得距离或欧几里得度量，是欧几里得空间中两点间“普通”（即直线）距离。

5. 实践：用 word2vec 提取维基百科特征

如何实验 Word Embedding 的强大能力呢？我们要找到训练素材，维基百科包含的自然科学知识覆盖面相对最广，可以很好地观察一个词的相关词，所以我们选用它做基本的素材。维基百科有很多版本，我们只采用了文字部分。

英特找到了目前最新的地址，你也可以照网址格式获取当前最新的版本。

https://dumps.wikimedia.org/zhwiki/20161001/zhwiki-20161001-pages-articles-multistream.xml.bz2

解压得到一个 XML 文件。我们观察下这个 XML 文件，只需要其中文字部分，使用 gensim 的 WikiCorpus 来读取 text。

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import logging
import os.path
import sys

from gensim.corpora import WikiCorpus
#run python process_wiki.py ../data/zhwiki-latest-pages-articles.xml.bz2 wiki.zh.text
if __name__ == '__main__':
    program = os.path.basename(sys.argv[0])
    logger = logging.getLogger(program)

    logging.basicConfig(format='%(asctime)s: %(levelname)s: %(message)s')
    logging.root.setLevel(level=logging.INFO)
    logger.info("running %s" % ' '.join(sys.argv))

    # check and process input arguments
    if len(sys.argv) < 3:
```

```
        print globals()['__doc__'] % locals()
        sys.exit(1)
    inp, outp = sys.argv[1:3]
    space = " "
    i = 0

    output = open(outp, 'w')
    wiki = WikiCorpus(inp, lemmatize=False, dictionary={})
    for text in wiki.get_texts():
        output.write(space.join(text) + "\n")
        i = i + 1
        if (i % 10000 == 0):
            logger.info("Saved " + str(i) + " articles")

    output.close()
    logger.info("Finished Saved " + str(i) + " articles")
```

读取出来的文字里有很多是繁体，也要记得处理，否则 word2vec 会当成两个词，影响最后的结果。

去了停词后，我们用以下代码来计算词嵌入模型。

```
model = models.Word2Vec(bigram_transformer[line_words],
size=feature_size, window=content_window, iter=iter,
min_count=freq_min_count,negative=negative,
workers=multiprocessing.cpu_count())
```

设置参数如下。

feature_size = 500 确定输出向量长度为 500 维。

content_window = 5 设定多少个词为一组进行计算。

freq_min_count = 3 只计算出现 3 次以上的词。

```
# 模型参数
feature_size = 500
```

```
content_window = 5
freq_min_count = 3
# threads_num = 4
negative = 3   #best 采样使用 hierarchical softmax 方法(负采样，对常见词有利)，不使用 negative sampling 方法(对罕见词有利)。
iter = 20
```

这样我们得到一个 500 维的基于 wiki 文档的模型，这个模型有 8GB，它在发现自然科学、社会科学的关联词中发挥了很重要的作用，也是语义网路的建立基础。

用这个 word2vec 的语义库来试试吧，先来个简单点的例子：汽车。从上到下列出依次相关的词。

```
请输入想测试的单词：汽车
2016-12-15 15:47:56,909: INFO: precomputing L2-norms of word weight vectors
轿车    0.615218341351
摩托车    0.575742840767
卡车    0.531364798546
跑车    0.520666182041
车辆    0.517995595932
自行车    0.516170024872
```

我们再来看看其他的例子。

```
请输入想测试的单词：量子
量子_纠缠    0.523558497429
量子_计算机    0.487917423248
粒子    0.467467755079
量子_谐振子    0.461190015078
凝聚态    0.454765617847
准_粒子    0.452007174492
量子_测量    0.448817074299
```

其中有些词-词的构成，这是我们启用了 word2vec 一个词组选项。可以根据需要选择打开或关闭。

也可以试试动词。

```
请输入想测试的单词：喜爱
喜欢    0.58508503437
喜好    0.563188910484
热爱    0.552253365517
爱好    0.526570439339
受欢迎    0.520372867584
观众_喜爱    0.497818797827
偏爱    0.484890878201
```

试试人名。

```
请输入想测试的单词：马云
李开复    0.532912015915
王健林    0.523721218109
俞敏洪    0.523662447929
马化腾    0.519190609455
慕辛    0.502838909626
郭_广昌    0.488249659538
丁磊    0.477871596813
许家_印    0.473230659962
```

由于计算词-词的距离特性，你可以更改word2vec的参数让其计算距离最远的词，这样可以得到通常意义上的反义词，如果你输入人名时，你会得到两个完全不相干的人，多试试，会很有趣！训练好的 word2vec 模型已经上传至百度云盘，http://pan.baidu.com/s/1bpbzt6n ，密码 62i2。请注意模型文件一共 3 个，共 8.1GB，最好在 16GB 内存以上的机器上加载，内存不够可能导致加载失败。

2.3.2 句子特征表示

英特在学习完词特征的表示后，再接再厉，学习了句子级别的特征表示，句子的特征表示分为传统方法 VSM 以及基于神经网络表示法的高级方法，下面先来看看 VSM。

1. 向量空间模型（VSM）

句子、段落和文章，我们都可以把它们视为是词的序列，因此在很多场景下可以用统一的方式来进行表示。当然，在涉及句法结构分析时，基本是以句子为单位的，这里暂时不考虑这种情况。作为词的序列，我们该如何去表示它们呢？理想情况下当然是希望词的顺序啊、语义啊、语法结构等都能够表达出来，但如果要将这些都反映出来，所使用的特征会比较复杂，光是语义和语法结构就够我们喝一壶了。

那么是否有一个相对简单的表示方法，在数学意义上比较简单，而表示逻辑意义又有可解释性呢？这里就要请我们的向量空间模型（Vector Space Model，VSM）出场了，VSM 经常和词袋模型（Bag of Words，BOW）联系在一起。

这里简单介绍下 BOW 的思想：BOW 假想我们有一个袋子，每遇到一个词就将其丢进袋子中，直到处理完所有文本。在此基础上，VSM 要求用一个向量来表示各个文档，这个向量的长度要与词袋中不同的词的数量一致。比如说我们有下面两段文本（案例引自维基百科）。

（1）John likes to watch movies. Mary likes movies too.

（2）John also likes to watch football games.

那么我们得到的词袋中包含的词就是 [John, likes, to, watch, movies, also, football, games, Mary, too]共 10 个词，这样两段文本可以分别表示为：

（1）[1, 2, 1, 1, 2, 0, 0, 0, 1, 1]

（2）[1, 1, 1, 1, 0, 1, 1, 1, 0, 0]

如上所示，不同位置的值为对应的词在文本中的权重，这里使用的是词频。这样两段文本之间的相似程度可以简单地通过算点积（dot product）来得到（如图 2-3 所示）。

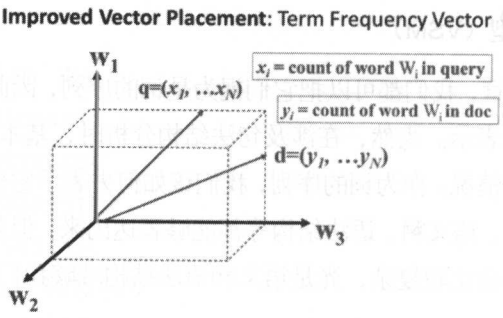


图 2-3　点积计算向量相似度

当然在实际中，我们希望不同维度的值是能反映对应的词在文本中的“重要程度”的，而直接使用词频并不能达到这个目的，事实上有一些常用的词会在很多的文本里都具有较高的频次，但它们本身并不包含重要的信息，我们希望抑制这些常用词，于是逆文档频率（IDF）被发明出来专门做这件事情。

TF-IDF 实际上是：TF × IDF，TF 词频（Term Frequency），IDF 逆向文件频率（Inverse Document Frequency）。TF 表示词条在文档 d 中出现的频率。IDF 的主要思想是：如果包含词条 t 的文档越少，也就是 n 越小，IDF 越大，则说明词条 t 具有很好的类别区分能力。

TF-IDF 的不足在于，其中的 IDF 表示一种试图抑制噪声的加权，并且单纯地认为文本频率小的单词就越重要，文本频率大的单词就越无用，显然这并不是完全正确的。IDF 的简单结构并不能有效地反映单词的重要程度和特征词的分布情况，使其无法很好地完成对权值调整的功能，所以 TF-IDF 法的精度并不是很高。

——来源：维基百科

VSM 存在两个问题，其中一个是其特征表示往往会很稀疏(想象一个包含了 100,000 个不同词汇的文本集合)，解决这个问题的方法之一是将一些文档频率很高的词去除，因为这样的词不能为文本与文本之间的区分性做贡献，此方法能有效地降低向量的维度并保留有效的信息。

VSM 的另外一个问题是缺乏语义信息。仍旧以上面两个句子为例，它们表达的

意义是不一样的，但在 VSM 中两者的表示会一模一样。

这个问题的解决办法之一是使用 n-gram 而非单词来作为基本单元，比如将 n 设为 2，上述两句话得到的词袋如下。

```
[
    "John likes",
    "likes to",
    "to watch",
    "watch movies",
    "Mary likes",
    "likes movies",
"movies too",
"John also",
"also likes",
"watch football",
"football games"
]
```

（1）John likes to watch movies. Mary likes movies too.

（2）John also likes to watch football games.

相对应的，两个句子可以表示为：

（1）[1, 2, 2, 1, 1, 1, 1, 0, 0, 0, 0]

（2）[0, 2, 2, 0, 0, 0, 0, 1, 1, 1, 1]

后续可以根据需要在向量空间下计算他们的距离等关键值。

除了 n-gram 外，还有其他的句子级别的表示吗？有的，这也是我们要介绍的第二种句子的表示方法：神经网络的句子表示法。这种方法可以分为基于 RNN 的表示方法和基于 CNN 的表示方法。

2. 神经网络的句子表示

神经网络的句子表示简单来说就是使用神经网络压缩表示句子信息。基于 RNN

的表示方法比较简单，即将句子中的词逐个输入到模型中，结束时取隐藏层的输出即可。这是因为 RNN（这里的 RNN 泛指 LSTM 及其之后的变形）隐藏层的输出是由之前的“记忆”和当前的输入计算得到的，可以简单地认为是“整个句子的记忆”，也就是一个句子的特征表示了（如图 2-4 所示）。

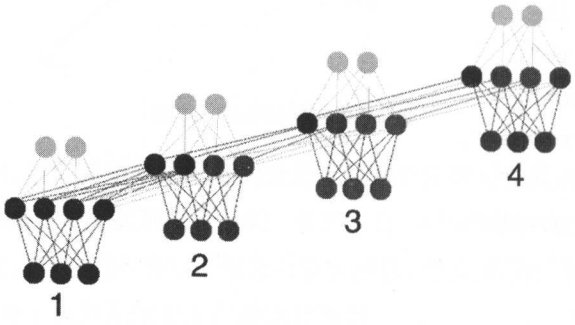


图 2-4　LSTM 句子表示示意图

基于 CNN 的表示方法则是将句子中的词的向量表示拼接成一个矩阵，然后在上面进行卷积，最后得到一个向量表示（如图 2-5 所示）。

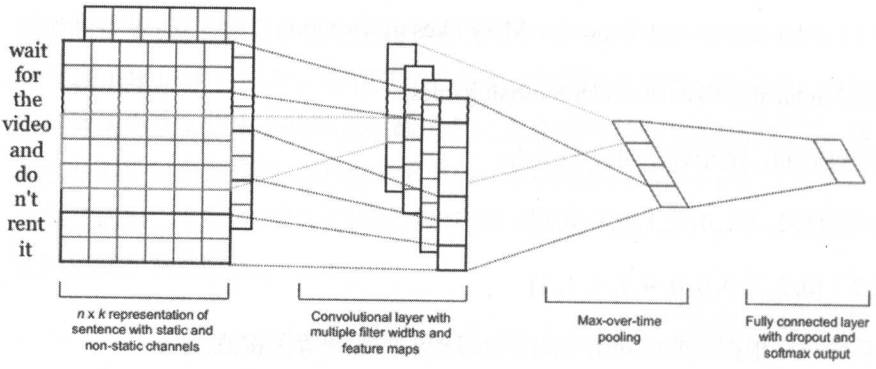


Figure 1: Model architecture with two channels for an example sentence.

图 2-5　CNN 句子表示示意图

CNN 或 RNN 都能够记录词在句子中的顺序，得到的句子嵌入（Sentence Embedding）同词嵌入一样，可以进行相似性的对比，语义相近的句子，其向量表示在空间中也会比较接近（如图 2-6 所示）。

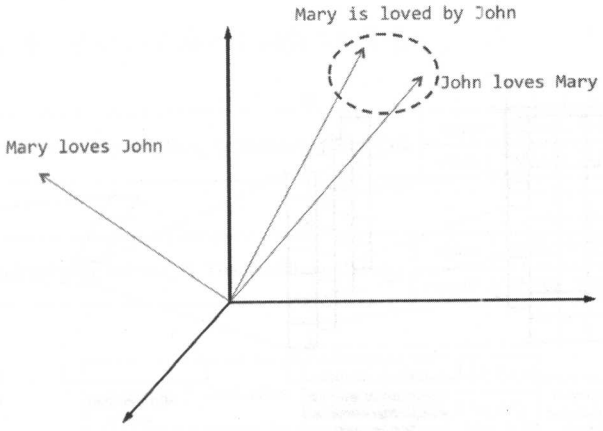


图 2-6 句子相似度

如果能像词嵌入那样得到句子嵌入，可不可以同样得到段落嵌入（Paragraph Embedding）乃至文档嵌入（Document Embedding）呢？对于前者，Tomas Mikolov 在 2014 年提出了“Paragraph Vector”，后者似乎还没见到过大规模的应用。在 Mikolov 的实验中，Paragraph Vector 在情感分析和信息检索两个任务上都取得了比其他模型更好的结果。不过和词嵌入一样，句子嵌入和段落嵌入的可解释性仍然存在问题。

3. 卷积神经网络（CNN）特征提取

英特在用邮件分类时，已经试过 CNN 卷积网路提取特征。那 CNN 为什么比原有的特征提取要好呢？原因在于基于卷积神经网络（CNN）来做文本分类，可以利用所包含的词的顺序信息。当然目前也有其他的方法可以利用词的顺序，而 CNN 独有的卷积且分层的概念可以保留更多更重要的信息。

图 2-7 是 CNN 模型的一个实现，共分四层，第一层是词向量层。Doc 中的每个词，都将其映射到词向量空间，假设词向量为 k 维，则 n 个词映射后，相当于生成一张 $n \times k$ 维的图像。针对词向量的映射有很多做法，你可以用 word2vec 提取特征，也可以用 One-Hot 表示法；第二层是卷积层。多个卷积核作用词向量层，不同核生成不同的 feature map；第三层是 pooling 层。这里做的是 Max Pooling，取每个 feature map 的最大值，这样操作可以处理长文档，因为第三层输出只依赖于滤波器的个数；第四层是一个全连接的 softmax 层，输出是每个类别的概率，如果我们不接 softmax 层，可以直接使用 Max Pooling 输出数据作为整个句子或文章的特征表达，用于下一个模

型的输入。

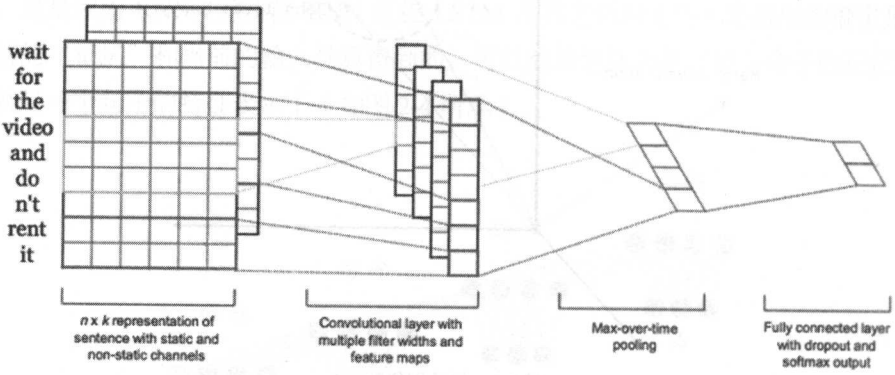


图 2-7　CNN 提取句子信息模型结构

这里谈到的原理也是英特用 CNN 文本提取开源库的基本原理，读者可以对照理解。

4．结合 CNN 和 RNN 的特征提取

分别学习完 CNN 提取文本和 RNN 提取文本后，英特延伸思考了一下，既然 CNN 与 RNN 在提取文本上都有各自的优缺点，如果将 CNN 和 RNN 合并成一个网络，既结合了两者的优点，也屏蔽了两者的缺点，该是一件多么完美的事。

如果要将 CNN 和 RNN 结合，首先要考虑的是先用 RNN 还是先用 CNN。英特仔细考察了两种网络的特性：RNN 在处理时序信息时有独特的优势，而语言（文本）天然具有时序特征，如果我们先用 CNN 进行卷积操作的话，那么时序特征有可能就无法完整地保留。基于这点考虑，开始时我们使用了 RNN 提取文本初始信息。为了更多地提取信息，我们在输入层后使用了一个双向的 LSTM 层——BiLSTM。

在 LSTM 层的下一层我们选用时序的包装器，目标是在时序处理上进行压缩。再往下，为了继续实现提取强特征（即决定句子意义的最大特征）的目标（该目标和池化层的目标意义非常相像），我们选用 Max Pooling 层做最大池化工作，目标是在句子的级别上提取最大化特征。在这里我们把参数 pool_size 选为 5，即每 5 个向量提取一个最大向量，最后我们将二维的输出拉平拉成一维输出，最终输出一个只有一个单元的层，这里做的是二分类，可以结合自己的应用选择多分类应用，或者提取

6592 特征层用于下一步处理的输入（如图 2-8 所示）。

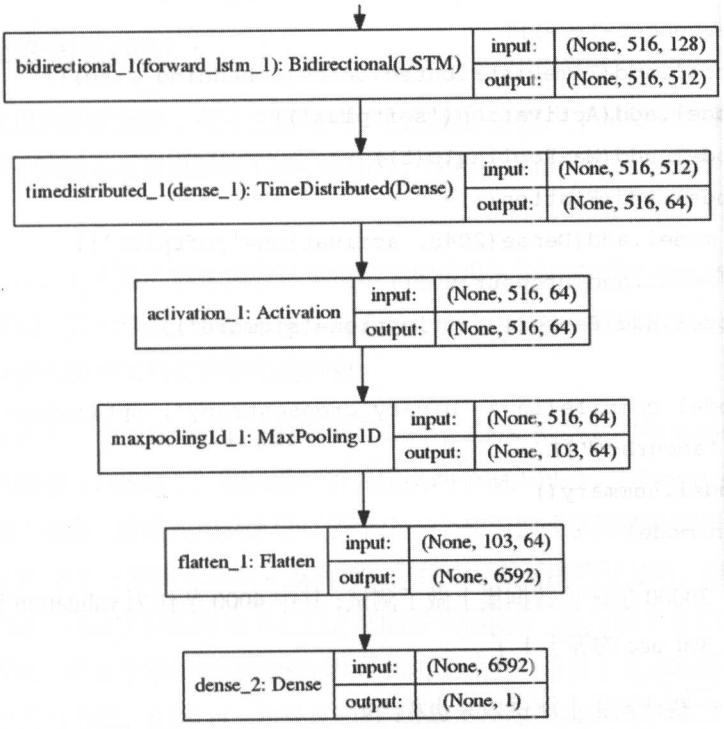


图 2-8　CNN+RNN 句子特征提取算法结构图

我们根据图 2-8 特征提取结构编写代码如下：

```
    def text_feature_extract_model1(embedding_size=128,
hidden_size=256):
        '''
        this is a model use normal Bi-LSTM and maxpooling extract
feature
        :return:
        '''
        model = Sequential()
        model.add(Embedding(input_dim=max_features,
                            output_dim=embedding_size,
```

```
                    input_length=max_seq))
        model.add(Bidirectional(LSTM(hidden_size,
return_sequences=True)))
        model.add(TimeDistributed(Dense(embedding_size)))
        model.add(Activation('softplus'))
        model.add(MaxPooling1D(5))
        model.add(Flatten())
        # model.add(Dense(2048, activation='softplus'))
        # model.add(Dropout(0.2))
        model.add(Dense(1, activation='sigmoid'))

        model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
        model.summary()
    return model
```

英特在20000条评论数据集上做了测试，其中4000条作为validation集合，训练2 epoch后，val_acc约等于1了。

在另外一些评论集上测试效果也相当好。

看看下面的例子。

这个货很好，很流畅 [1.62172219e-05]

这个东西真好吃， [1.65377696e-05]

服务太糟糕,味道差 [1.]

你他妈的是个傻X [1.]

这个贴花的款式好看 [1.76498161e-05]

看着不错，生产日期也是新的是16年12月份的，就是有点小贵 [1.59666997e-05]

一股淡淡的腥味。每次喝完都会吃一口白糖 [1.]

还没喝，不过，看着应该不错哟 [1.52662833e-05]

用来看电视还是不错的，就是有些大打字不习惯，要是可以换输入法就好了！ [1.]

嗯，中间出了点小问题已经联系苹果客服解决了，打游戏也没有卡顿，总体来讲还不错吧！ [1.52281245e-05]

```
下软件下的多的时候死了一回机，强制重启之后就恢复了。 [ 1.]
东西用着还可以很流畅！ [ 1.59881820e-05]
```

2.3.3 深度语义模型

在前文说的词的特征、句子的特征中，已经有部分使用了深度网络来提取特征，特别是 CNN 的加入或者说深层 CNN 的加入，使得深度语义特征的提取有了长足的进步。

在项目中，有大量的需求需要计算文本的相似度，以前我们使用 VSM 文本表示法在向量空间计算相似度，现在如果我们能将文本的语义特征提取出来，那么两个文本间的计算就变成语义特征的相似度计算。

下面我们看一个微软 2014 年的案例，即利用 CNN 网络做的语义相似度——深度语义相似模型（DSSM）。如何做？首先定义两个输入维度，用 word sequence 表示这两个输入维度，这是先用文本特征表示出一个基于词或字的特征向量（中文处理方式）X 和 Y，XY 分别输入到两个卷积神经网络中，做深层特征抽取，得到两个 128 维的特征向量；用这个特征向量求它们之间的余弦相似度，求得两个文档或两段文字之间相似程度。而这个求相似度的模型就是利用了深度网络对于文本的特征提取。微软的命名也很有意思，f()和 g() 如果是一般 DNN 网络就叫 DSSM，如果是 CNN 网络则命名为 C-DSSM。

图 2-9 是一个具体的原理图，表现了计算相似语义空间的方法，而图 2-10 给出了一个基于此方法的 CNN 网络细节图和原理说明，左侧网络是一个典型的卷积网络，可以清晰地看到卷积层和池化层，我们需要注意这是非分类模型，所以后一层输出的是 128 维的特征向量，用于做最后的相似度比较。

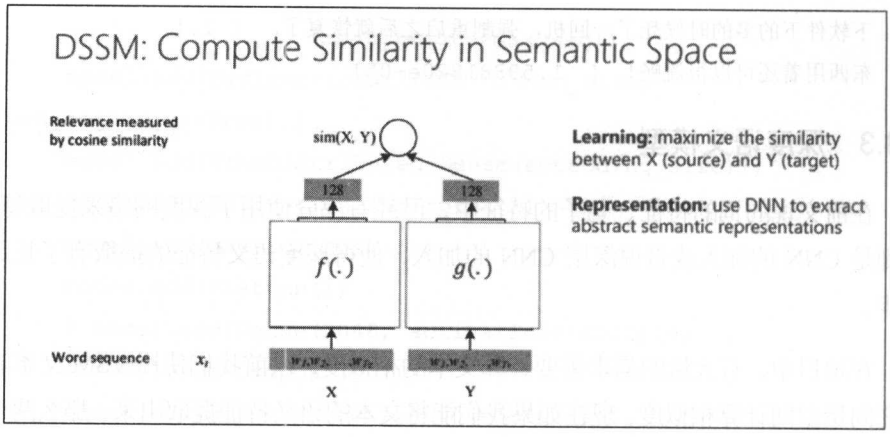


图 2-9　DSSM 模型

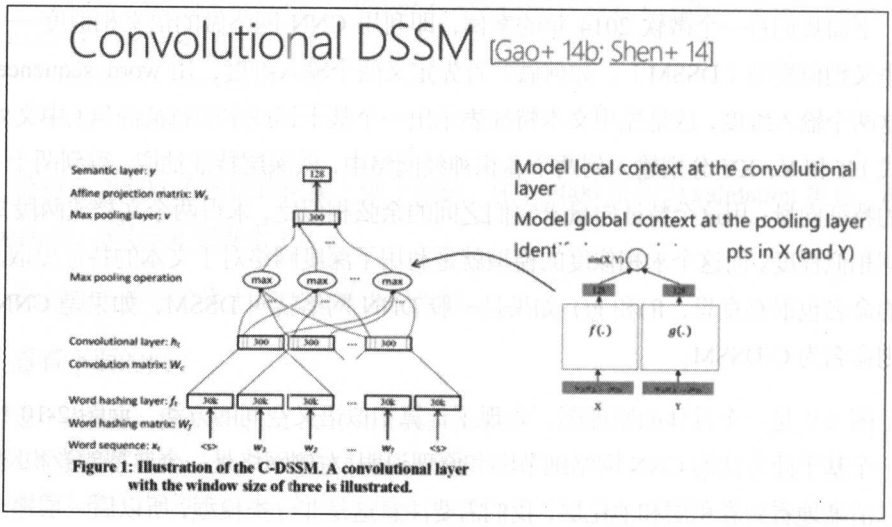


图 2-10　DSSM 模型部分细节

总结一下，深度学习曾被认为不适合用来做语义理解，主要是因为词语之间的相似程度与其含义的相似程度并无太大关系，词表的出现一定程度上解决了这个问题。现在，深度学习在语义理解上的障碍已经基本不存在了，微软此次提供了深度语义做相似度提取的思路，也将 CNN 作为文本特征提取融汇到真正的应用中（比对相似性）。2015 年 Google 也发了一篇论文做问题和答案的相似程度匹配，它们的主要思想都是一致的。

3 做一个对话机器人

完成了一些文本的项目后，英特的团队感觉对于文本类的应用做得更加得心应手了。这时候公司交给英特一个非常具有前瞻性的项目：智能对话项目。

人类其实从很早以前就开始追求人类和机器之间的对话，早先科学家研发的机器在和人对话时都是采用规则性的回复，比如人提问后，计算机从数据库中找出相关的答案来回复。这种规则性的一对一匹配有很多限制。机器只知道问什么答什么，却不知道举一反三，比如你问它："今天天气怎么样？"它会机械地把今天的天气告诉你。这不像人与人之间的对话，人是有各种反应的，这类反应的产生是基于人的知识结构和对话场景的。

那么，你觉得这类机器是否真的具有智能了？图灵测试是这样判断机器人是否具有智能的：测试中，一个正常人将尝试通过一连串的问答，把被试的机器与人类区分开来。一般来说，如果正常人无法分辨和自己聊天的是人还是机器人的时候，机器人就算通过测试了。

图灵测试的关键之处在于，没有定义"思维/意识"。只是将机器人作为黑盒，观察输入和输出是否达标。所以说它从一开始就绕开了"机器能思考吗？"这样的问题，而是把它替换成另外一个更具操作性的问题——"机器能做我们这些思考者所做的事吗？"。大家注意这两者其实完全不是一个层次的问题。

然而，“机器能思考吗？”和“机器能做我们这些思考者所做的事吗？”这两个问题真的可以相互替代吗？

比如说，机器能够写诗，甚至比许多资质平庸的人写出的诗更像样子。如果我们人为拟定一套标准，来为机器和人写的诗打分，那么完全有可能设计出一台能够赢过绝大多数诗人的写诗机器。但这真的和人类理解并欣赏一首诗是一回事吗？再比如，人工智能在国际象棋、围棋领域已经比人类更强大，但这真正和人类思考如何下棋是一样的吗[1]？

世界上有这么一个关于图灵测试的奖项——“勒布纳奖”，颁给擅长模仿人类真实对话场景的机器人。然而，这个奖项大多数的获得者都没有看上去那样智能。比如一个人问一台机器“你有多爱我？”，如果它想通过图灵测试，它就不停地顾左右而言他，比如回答“你觉得呢？”事实上大多数问题都可以用反问去替代，说白了这些仅仅是一些对话技巧。而获胜者并没有真正理解“你有多爱我？”这样的问题。

这里有句话，希望大家记住：**人工智能的真实使命是塑造智能，而非去刻意打造为了通过某类随机测试的“专业”程序。**

所幸到今天为止，很多学者都意识到了图灵测试的局限性，如果我们要发明人工智能，就要真正清楚地定义人工智能。同样如果我们要做智能对话，我们也要清晰地定义智能对话。

在2013年的一次国际会议上，来自多伦多大学的计算机科学家[2]发表了一篇论文，对“图灵测试”提出了批评。他认为类似这样的人机博弈其实并不能真正反映机器的智能水平。对于人工智能来说，真正构成挑战的是这样的问题：

镇上的议员们拒绝给愤怒的游行者提供游行许可——“因为他们担心会发生暴力行为”——是谁在担心暴力行为？

---

1 人工智能确实比人类强大了，甚至德州扑克 AI 也证明强于人类了，但这和大脑处理下棋或扑克时的机制不是完全一样的。大脑善于处理多种任务，而无论是围棋还是德州扑克 AI 都只是单一任务。

2 赫科特·勒维克。

A. 镇上的议员们

B. 愤怒的游行者

类似这样的问题，机器有没有可能找到正确的答案？要判断“他”究竟指代谁，需要的不是语法书或者百科辞典，而是常识。人工智能如何能够理解一个人会在什么情况下“担心”？这些问题涉及人类语言和社会交往的本质，以及对话的前后语境。这些本质其实是一种规则，而这种规则是在不停变化的。正是在这些方面，目前人工智能还无法与人类相比。

这意味着，制造一台能与人类下棋的机器人很容易[1]，但想要制造一台能理解人类语言的机器人却很难。

为了更好地理解机器对话，英特将现有的对话技术进行总结并画出流程图（见图3-1），这里面涉及的逻辑和模块较多，英特是从模拟人类对话的第一步，即理解人类的语言开始的，当然要做到完全理解人类的语言在目前来讲也不太可能。对于机器人来说，无论何种用途的机器人，首要需要解决的就是理解人类说了些什么，而除了命令句式以外，理解人类说什么就是理解人类提出的各种问题。请移步下一节来看看如何理解人类的提问。

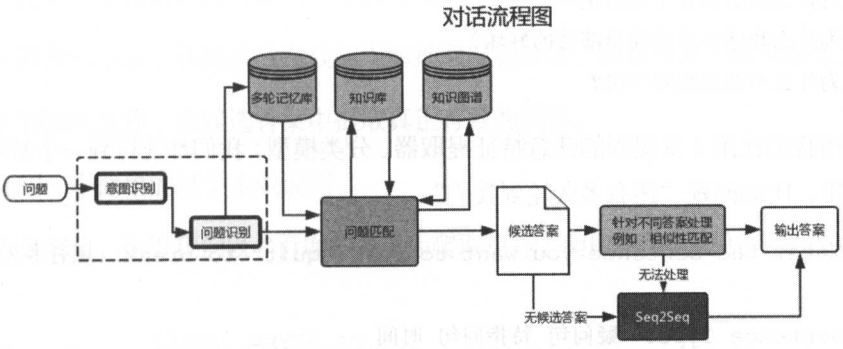


图 3-1 对话流程图

1 此处是为了行文的连贯性，并不是贬低 AlphaGo 的成就，特此注明。

3.1 理解人类提问

英特调查后发现对于中文问题来说，无非可以分成以下两类：疑问句和反问句。对于反问句当然没什么好说的，我们来重点看看疑问句。可以分为是非问句、正反问句、特指问句、选择问句，其中特指问句又可以分为人、原因、地点、时间、意见、数量、方式和其余的实体。

对于问题来说，人类也需要首先对句子做一个判断，拿特指问题来说，需要判断到底是问什么？接着将每个问题做一个初步的定位，缩小回答时的搜索范围，最后从知识体系和场景中取得答案。

英特团队按照句式结构找了些例句放进去，为后一步的句式分类准备好训练集。

比如在“特指问句_时间”里放入了如下例句。

中国第一部宪法颁布的时间？
哪天你有空？
演唱会是哪天？

又比如“特指问句_原因”里放入如下例句。

为什么人的面容千差万别？
为什么我感觉不到演员演技的好坏？
为什么不要空腹喝牛奶？

用我们在第 2 章提到的任意特征提取器、分类模型，我们都能得到一个基本准确的输出，比如问题“还有多久轮到我们”。[1]

```
Enter the sentence you want to test("quit" to break):还有多久轮到
我们
Sentence Type:  疑问句_特指问句_时间 -- <type 'str'>
How much:
疑问句_特指问句_时间 --> 0.568171744082
陈述句_转折复句 --> 0.0833727296382
```

1 这里特意没有写问号，以验证分类器的准确性。

```
陈述句_目的复句 --> 0.0702280010834
陈述句_时间复句 --> 0.0467500544003
陈述句_连锁复句 --> 0.0389512385469
疑问句_特指问句_地点 --> 0.0360868190755
陈述句_因果复句 --> 0.023920374049
疑问句_选择问句 --> 0.0149039847156
疑问句_特指问句_意见 --> 2.89120579329e-19
脏话_增强语气 --> -0.00288297881955
脏话_恶意脏话 --> -0.00377381341484
```

3.2 答案的抽取和选择

在答案的提取阶段，一般的对话像常见的智能对话助手 Siri、小冰等，都是有对应的问题答案组（QA）的，这种 QA 数量一般都接近百万级了。而在现实工作中，没有能力和精力人工组建QA怎么办？这个时候我们可以使用互联网的信息——利用爬虫爬取。

大体过程是这样的：

（1）定义一个爬虫，针对某些问题的特点爬取候选答案。

（2）答案的抽取。从离线或在线知识库抽取候选答案，候选答案一般有多条。

（3）答案的选择。从候选答案中提取真正有效的回答。

下面用一个简单的例子来做说明。

首先是爬虫，这里我们以“百度知道”为爬取目标，爬取相关问题及答案的前几名。

```
def getAnswerfromZhiDao(question):
    """
    Scrap answers from ZhiDao(百度知道 )
    :param question:
    :return:
    """
```

```
        tic = time.time()
        global zhidaoHeader
        URL = ZHIDAO + "/index?rn=10&word=" + question
        # print(URL)
        Answer = []
        http = httplib2.Http()
        # 声明一个 CookieJar 对象实例来保存 cookie
        cookie = cookielib.CookieJar()
        opener =
urllib2.build_opener(urllib2.HTTPCookieProcessor(cookie))
        req = urllib2.Request(URL)
        response = opener.open(req)
        zhidaoHeader['Cookie'] = response.headers.dict['set-cookie']
    response, content = http.request(URL, 'GET',
headers=zhidaoHeader)
```

上面的代码是整个爬虫的 http 请求 header 头组装及爬取 url 的组装。

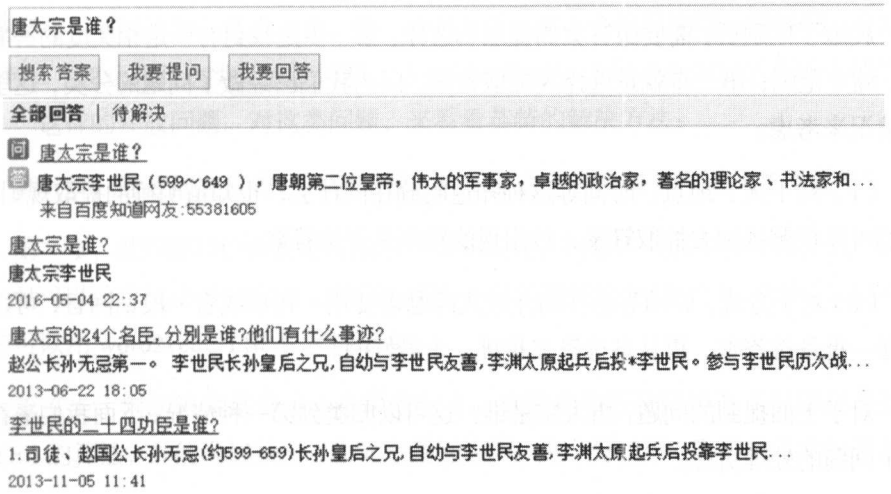
```
ZHIDAO = 'http://zhidao.baidu.com'
URL = ZHIDAO + "/index?rn=10&word=" + question
```

这里参数 rn=10 也就是要返回 10 条答案。

比如用户提问：唐太宗是谁?

组装后 URL = https://zhidao.baidu.com/index?rn=10&word=唐太宗是谁?

利用组装的 URL 返回答案条目如下。




```
        search_result_list =
etree.HTML(content.lower()).xpath("//div[@class='slist']/p/a")
    # time1 = time.time()
    limit_num = 3
    for index in range(min(len(search_result_list), limit_num)):
        url = search_result_list[index].attrib['href']
        url = ZHIDAO + url
        # print(url)
        response, tar_page = http.request(url, 'GET',
headers=zhidaoHeader)

        # 判断是否做了重定向 301, 302
        if response.previous is not None:
            if response.previous['status'][0] == '3':
                url = response.previous['location']
```

上一段程序先爬取答案的一级页面，也就是上面截图的页面，然后再分别爬取前三个答案的详细页面，以得到具体的答案。也可以控制爬取详细答案的数量：limit_num = 3。在爬取“百度知道”内容时需要注意重定向问题，以区分处理各种网站。

爬取了问题后，需要做答案的提取和选择，第一步是找到有一定相关度的答案候选、缩小范围；第二步就是选择并提取答案。因为我们已经做了问题的分类，这里要分情况来考虑。

（1）对于人、地点、时间等这种知道明确在问什么，也知道明确的提取规则的，我们可以按照规则去抽取答案，按出现的频率来评价答案。

（2）对于方式、原因等这样的开放式问题需要用一句话或者一段话回答，可以先选择一批备选答案，再从备选答案种挑一个最好的。

对于上面提到的问题：唐太宗是谁？这可以归类到第一种情况。下面我们来看看这个问题的处理流程。

第一步，通过问题分类、问题理解，找到了这个答案属于特指问题“人”。

第二步，针对这类问题，英特用了个小技巧，非常简单但有效：如果我们确认要找的是人、人名等，可以在备选答案中选择表示人的单词，比如名词、代词，再计算它出现的频率，比如“李世民”这个词在答案中出现了 10 多次，就可以将它提取出来反馈给用户。

```
if tar == 'who':
 for i in words:
    if i.flag == 'nr' and i.word not in question:  # 人名
       humWord.append(i.word)
 return getTopWord(humWord, 1)
```

问数量[1]的、问时间的、问地点的都可以照此处理。当然如果这里单独用了词频还是有问题，我们将范围缩小一些找词频就能更准确：比如将“唐太宗是谁？”这个问题在答案里做相似性匹配，找到匹配度很高的候选答案，再用以上的小技巧计算词频，当然在计算词频时，我们可以看看比如“李世民”这个词出现在句中的位置，这能帮助我们进一步精确地抽取答案。

做到这一步，英特解决了简单问句的回答，特别是针对时间、人物、地点等这种

1 数量的处理时会去先找度量单位。

回答只有一个词的问句特别有效。但对话中大量出现上文提到的第二种情况，诸如问“如何”“怎么样”这类开放性问题，该如何处理呢？比如“唐太宗是个什么样的皇帝？”这样的开放问题。对这类问题，来看看英特的解决方法：

第一步，通过问题分类、问题理解，找到了这个答案属于特指问题“意见”。

第二步，求问题和检索答案段落的文本相似性。相似性有很多种处理方式，思路之一是可以简单地估计问题的关键词占所有问题关键词的比例（比如：唐太宗、皇帝），或者可以用 word2vec、LSI 来求相似度。

第三步，判断答案是否由一个段落蕴含。这里我们假设已经有了识别文本蕴含（RTE）的算法，并能准确找出“唐太宗勤勉治国，是个好皇帝。”是被蕴含的答案。

我们将答案提取打分的伪代码总结如下：

```
    ScoreAnswers (String[] answers, String[] passages,
AnalyzedQuestion aq)
    scoredAnswers←ø
    foreach answer(answers):
        //特征 1：段落和问题间的文本相似度

    textSim←calculateTextSimilarity(answer,passages,aq,question)
        //特征 2：判断一个蕴含问题
        entailed←recognizeEntailment(answer,passages, aq,question)
        //从特征值估算置信分值
        //通过增强置信分值来找到相似答案
        return scoredAnswers
```

通过相似度评分和蕴含评分得到答案的总体评分，然后再将答案排序输出第一个答案。

对于相似度评分无需过多描述，但是蕴含呢？英特查了很多资料后决定了蕴含关系的研发方案。

3.3 蕴含关系

蕴含关系，是为了评价从一段文字中得到的推论是否符合原文的本意，我们这里用蕴含关系来做答案中是否包含着问题的判断，其实就是求某种语义上的相似性或相关性。

下面举个例子。

T:第一次世界大战（简称一战；英语：World War I、WW I、Great War）是一场于1914年7月28日至1918年11月11日主要发生在欧洲，然而战火最终延烧至全球的战争，当时世界上大多数国家都被卷入这场战争，是人类史上第一场全球性规模的大型战争，史称"第一次世界大战"。

H:第一次世界大战的时间

label:1←这里标签为1，表示答案中蕴含问题。

T:第二十九届奥林匹克运动会（英语：the Games of the XXIX Olympiad；法语：les Jeux de la XXIXe Olympiade），又称2008年夏季奥运会或北京奥运会，于2008年8月8日至24日在中华人民共和国首都北京举行。

H:东京奥林匹克运动会的举办时间

label:0 ←这里标签为0，表示答案中没有蕴含问题。

从例子可以看出，求蕴含关系就是求一个相似度，但还不完全像求相似度，蕴含关系中，选择哪些特征才是这个算法在问答中应用的重点，只要把特征选出扔到SVM分类器中就可以做训练了。

一般提取哪些特征出来呢？我们先人工选择特征并提取。看看代码，除了词的频率和位置还可以提取下面这些特征（规律）：

```
features['word_overlap'] = len(extractor.overlap('word'))
 # hyp 与 text 中重复的 word
features['word_hyp_extra'] = len(extractor.hyp_extra('word'))
 # hyp 有 但 text 中没有的 word
features['ne_overlap'] = len(extractor.overlap('ne'))
# hyp 与 text 中重复的 ne
```

```
    features['ne_hyp_extra'] = len(extractor.hyp_extra('ne'))
    # hyp 有 但 text 中没有的 ne
    features['neg_txt'] = len(extractor.negwords &
extractor.text_words)
    # text 中的 否定词
    features['neg_hyp'] = len(extractor.negwords &
extractor.hyp_words)
    # hyp 中的 否定词
```

ne 指的是命名实体（Named Entity）[1]，其中 hyp 指的是问题，大家观察蕴含的示例代码可以看出，英特在项目进行中发现光是名词实体不足以提取到完整特征，于是将时间名词、成语、状态词都加入到 ne 范畴中，提取规则同上保持不变。

当然如果你不想用人工方式提取答案和问题的特征，仍然可以用在第 2 章我们提到的 CNN+RNN 方式提取特征，而这种提取方式可以稍作变化，将词性作为输入加到 Word Embedding 层后。

3.4 生成式对话模型（Generative Model）

前面的这类方法可以总结为检索式方法，其思路是从一个已知的大知识数据库[2]中搜索并组合出相应的答案，这种搜索方式可以伴随一些预定好的规则，比如：{who} 想要糖果？回答：我想要{ pronoun }。 这个{ pronoun }可以从数据库找出相应的代词填入。而这种回答的规则可以用一种通用的 XML 文件来描述，我们称它为 AIML2.0。

另外还需要拥有初级的逻辑推理能力，比如 IBM Watson ，而对于这类而言，我们除了需要制定相应的逻辑规则库以外，还需要事实库。在 Python 中我们可以选用 pyke 框架来管理相应的事实库与逻辑规则库。

以上无论是逻辑推理回答还是一般回答都可以称作检索式的或者规则式的方法。除此之外还有哪些方法可以给出答案呢？英特在调查中发现目前学界在 RNN(LSTM)

1 指识别文本中具有特定意义的实体，主要包括人名、地名、机构名、专有名词等。

2 这个数据库可以是离线的已整理好的库，也可以是在线的利用互联网信息提取的数据库。

上的突破让模型拥有了初级的‘学习’能力：基于神经网络序列的生成式对话模型。

生成式对话模型算法的概念就是让模型先看一些对话集，然后问它一句话，模型会通过从对话集学习的一些规律给你一个回答。简单来说根据你的上一句话和学习到的所有对话集规律生成一个个单词，这些单词如果意义是连贯的那就是一句话！而这正好契合了对话模型。

听上去很美好，而且一些大公司已做了很多工作，比如图 3-2 所示的 Google 对话集[1]：

Human: ***what is the purpose of life ?***
Machine: ***to serve the greater good .***
Human: ***what is the purpose of living ?***
Machine: ***to live forever .***
Human: ***what is the purpose of existence ?***
Machine: ***to find out what happens when we get***

图 3-2　Google 对话集示例

Google 的这个结果建立在训练了 3000 万条对话的基础上，取其中的 300 万条对话来做验证集，并且去掉了专业名词、数字和 URL 等。达到这种数据量级后，能够对某一垂直领域类的问题做自由度较高的机器回复。

Google 所用的生成式对话模型有哪些特点呢？下面来看看基于检索式模型和生成模型之间的区别，是不是能找到一些特点。

基于检索式模型（基于 AIML2.0）使用了预定义回复库和一种条件触发式方法来根据输入和语境做出合适的回复。这种触发式方法一般基于规则的表达式匹配，当然你也可以用机器学习分类器来处理这类触发。检索式模型的特点是它不会产生新文本，只是从固定集合中挑选一种回复，套用农夫山泉广告语：“我们不生产文本我们只是集合的搬运工”。

生成式对话模型不依赖于预定义回复库，从零开始生成新回复。生成式对话模型一般基于机器翻译中的 Seq2Seq 技术，但应用场景有较大差别；机器翻译的目标是：

1　论文链接 http://arxiv.org/pdf/1506.05869v1.pdf。

把一个输入“翻译”成一个输出“回复”。输入（翻译）再输出（回复）就是编码器（encoder）经信息编码后再经解码器（decoder）解码的过程，而在对话中输入人说的话再输出机器的回复（如图 3-3 所示）。

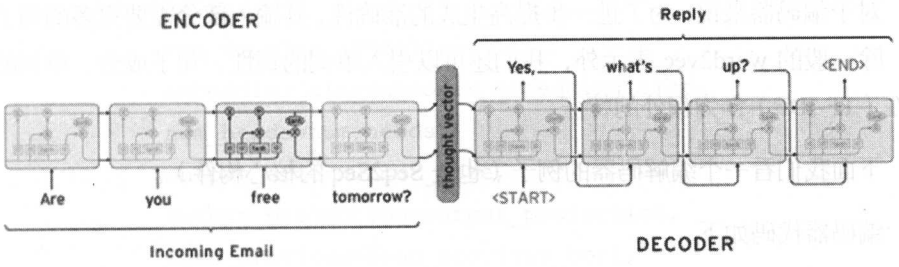


图 3-3 Seq2Seq 结构

编码器（encoder）是对于人的提问或对话的编码，可以看作是机器对问题的一种理解，而解码器（decoder）是机器对问题的一种回复。

是不是生成式对话模型就远比检索式要好呢？可以说目前为止，还只是各司其职，各自有不同的应用场景：

（1）检索式模型由于采用人工制作的回复库，基于检索式方法不会有语法错误，当然我们使用搜索引擎作为回复库，也很少有语法错误。然而使用回复库不能处理没出现过的情况，因为它们没有合适的预定义回复。同样，这些模型不能重新利用提上下文中的实体信息，如先前对话中提到过的名字。综上，检索式模型可以用在需要正确回答问题的场合，对答案的语法和准确性要求比较高。

（2）生成式对话模型从原理上讲更“聪明“些。它们可以重新提及输入中的实体并带给你一种正和你对话的感觉。然而，这类模型很可能会犯语法错误（特别是输入一个长句时），而且通常要求大量的训练数据。综上，生成式对话模型可以用在要求不那么精确的对话中。比如游戏的 NPC 交谈，比如一般的生活类对话场景。

虽然生成式对话模型有一些缺点，但毕竟是一个很好的方向，英特选择使用 TensorFlow 来实现这个结构。

前文提到，生成式对话模型是基于生成式机器翻译模型，而机器翻译模型用的就是 Seq2Seq（Sequece2Sequence）结构 。我们先来理解下 Seq2Seq 结构：Seq2Seq 由

编码器和解码器组成，输入的单词以序列化的方式传入编码器，最终得到表示一句话的上下文特征向量；解码器接收特征向量以及每次的输出单词，做序列化的解码，输入和输出以<EOS>终止。

对于编码器来说，为了进一步提高生成的准确性，其输入部分需要更多的句子特征，除一般的 word2vec 表示外，我们还可以引入单词的词性、句子成分、单词的重要性（TF-IDF）作为额外的特征。

下面我们看一个编解码器的例子（也是 Seq2Seq 的核心构件）。

编码器代码如下。

```
# Encoder
encoder_cell = rnn_util.MultiEmbeddingWrapper(
    cell,
    embedding_classes=num_encoder_symbols,
    embedding_size=encoder_embedding_size
)
encoder_outputs, encoder_state = rnn.rnn(
        encoder_cell, encoder_inputs, dtype=dtype)
```

解码器代码如下。

```
# Decoder.
output_size = None
if output_projection is None:
    cell = rnn_cell.OutputProjectionWrapper(cell,
num_decoder_symbols)
    output_size = num_decoder_symbols
.....

def decoder(feed_previous_bool):
    reuse = None if feed_previous_bool else True
    with
variable_scope.variable_scope(variable_scope.get_variable_scope(),
reuse=reuse) as scope:
```

```
        outputs, state = embedding_attention_decoder(
            decoder_inputs,
            encoder_state,
            attention_states,
            cell,
            num_decoder_symbols,
            embedding_size=decoder_embedding_size,
            num_heads=num_heads,
            output_size=output_size,
            output_projection=output_projection,
            feed_previous=feed_previous_bool,
            update_embedding_for_previous=False,
            initial_state_attention=initial_state_attention)
        state_list = [state]
        if nest.is_sequence(state):
            state_list = nest.flatten(state)
        return outputs + state_list

    outputs_and_state = control_flow_ops.cond(feed_previous, lambda:
decoder(True), lambda: decoder(False))
```

这里解释下解码器中的 Attention 层，Attention 层的思想也是来自于翻译领域，即前文出现的单词也可能出现在后面的回答中，比如说人名、地名等信息。Attention 层是在编码器隐藏层和解码器之间的网络结构，在某一时刻 t 时，接收解码器的隐藏层信息，生成当前时刻的加载到编码器隐藏层上的权重。

英特训练了 10 万条有关星巴克的微博信息，最终取得了不错的结果，我们先来看看中间的输出过程（如图 3-4 所示）。

```
you> 饥渴的复
>> 安 赏 盗 令 获 薛 a 宫 没 让 人 连 得 赏 留 春 井 揍 巾 都
you> 士大夫士大夫
>> 理 位 没 让 人 侄 句 腼 倒 偷 喜 宫 虾 道 思 声 思 广 节 途
you> bowie
>> 没 让 人 命 边 关 角 子 直 进 因 进 行 令 拒 多 从 新 在 他
you> 开机的开发建设的接口
>> 假 唤 感 按 得 忙 但 坠 得 下 才 感 按 得 嗜 没 让 人  瞎
you> 肌肤是看呆了发几
>> 非 奉 侄 好 得 直 生 杨 沃 心 关 赏 血 嘘 歹 王 歹 宫 内 把
you> 模拟试卷卡里的发卡
>> 他 解 宫 门 迟 刮 赶 心 角
you> 你看那
>> 没 让 人 宫 命 各 撤 缺 司 拔 伶 争 得 命 各 撤 缺 泄 露 险
vou>
```

图 3-4　经过 5 epoch 后的训练结果

图 3-4 是英特用和星巴克有关的微博数据训练了 5 epoch 后得到的结果。可以看出，在 5 epoch 时，模型还无法说出一个完整的句子，甚至都无法表达通顺的短语，但是模型在不停的学习中回答越来越准确，下面的图 3-5 是训练了 50 epoch 后的最终结果。

```
you> 你为啥会买星巴克
>> 星 巴 克 里 最 喜 欢 的 就 是 凯 撒 鸡 肉 卷 ， 爱 到 一 天
you> 鸡肉卷
>> 星 巴 克 的 红 豆 松 饼 和 鸡 肉 卷 ， 红 豆 松 饼 自 己 做
you>  小资情调
>> 【 买 了 六 年 中 杯 、 被 问 六 年 “ 你 确 定 吗 ” ？ 杭
you> 星巴克小资吗
>> 【 星 巴 克 不 想 只 靠 中 产 赚 钱  高 速 扩 张 至 三 四
you> 红杯
>> 星 巴 克 红 杯 来 啦 ！ 你 是 不 是 也 期 待 了 很 久 ？ 这
you> 圣诞节到啦
>> 星 巴 克 红 杯 来 啦 ！ 你 是 不 是 也 期 待 了 很 久 ？ 这
you> 是否你认为已经可以完美录
>> 星 巴 克 活 动 ！ 直 播 时 一 直 抽 奖 ！ （ 来 自 ） A P
you> fuck
>> 星 巴 克 红 杯 来 啦 ！ 你 是 不 是 也 期 待 了 很 久 ？ 这
you> 你确定要免费领取一杯星巴克吗
>> 【 买 了 六 年 中 杯 、 被 问 六 年 “ 你 确 定 吗 ” ？ 杭
you> 免费星巴克
>> 你 是 不 是 在 上 海 吃 了 ？
you> 星巴克免费
>> 我 也 想 去
vou>
```

图 3-5　经过 50 epoch 后的训练结果

英特对最终结果分析后发现仍然会存在一些问题：第一就是前文提到的会有一些语法错误；第二需要大量的训练数据。

第一个问题主要受限于现在的模型原理。目前暂时没有哪个模型或者衍生的模型能解决好。

再看第二点，如何获取大量的训练数据。凭借英特的经验，对于普适性的对话模型可以从两类途径获取：一是从电视剧中获取相关数据；二是从微博、QQ 聊天记录中获取相关数据。

对于一些专业性比较强的领域，就要求在本专业领域收集数据了，如上文提到的 Google 将自己 IT 服务部门的所有对话拿来训练。在任何稍微开放领域的应用上，比如像回复一封工作邮件，就超出了该模型现有的能力范围。但退一步来讲，仍旧可以利用模型建议和改正回复来“辅助”人类工作者；然后在这个过程中让模型学习人类真实的回复语句，不断更新出一个符合人类习惯的对话模型。

3.5 判断机器人说话的准确性

英特团队在做完以上工作后，发现机器说的话有些已经有成形的答案，还有些不通顺、不成句子。这涉及另外一个问题，我们如何评价模型呢？如何判断哪些模型的哪些调整有助于提高输出句子的通顺性或准确度？

英特浏览了相关的论文，发现针对这类问题有很多评价办法，最终英特选择了最早出现在翻译界的方法：BLEU 评测方法。

> BLEU（Bilingual Evaluation understudy）方法由 IBM 提出，这种方法认为如果机器翻译的译文越接近人工翻译结果，那么它的翻译质量越高。所以，评测关键就在于如何定义系统译文与参考译文之间的相似度。BLEU 采用的方式是比较并统计共现的 n-gram 个数，即统计同时出现在系统译文和参考译文中的 n-gram 的个数，最后把匹配到的 n-gram 的数目除以系统译文的单词数目，得到评测结果。

最开始提出的 BLEU 法虽然简单易行，但是它没有考虑到翻译的召回率。后来对 BLEU 做了修正，即首先计算出一个 n-gram 在一个句子中最大可能出现的次数 MaxRefCount(n-gram)，然后再和候选译文中的这个 n-gram 出现的次数比较，取它们之间最小值作为该 n-gram 的最终匹配个数。

3.6 智能对话的总结和思考

前面的工作可以总结为简单流程图（如图 3-6 所示）。

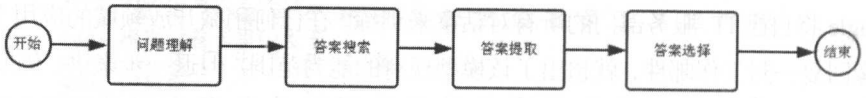


图 3-6 对话简单流程

该流程一共包括 4 个主要模块。正如本章开始时所分析的那样，最优先解决的是问答，与其说这是一个智能对话机器人不如说这其实就是信息检索和总结的过程，因为聊天毕竟不是一个人的事，它必然是一个交互的过程（程序与人的交互过程）。而解决交互过程最好的方法就是应用强化学习（reinforcement learning），我们会在后面的章节中对该算法做具体的实验说明，这里仅简单描述：强化学习是用来解决程序与环境的交互问题的，即让程序对当前所处的环境做出必要的反应。

假定我们站在机器的角度来考虑问题，所处的环境为聊天室，看到的全是我与对方的聊天记录，我们要做的就是在适当的时刻给出合适的回复，那么这里就需要做三件事：

（1）看懂聊天记录（state）；

（2）量化回复所用的语言（action）；

（3）针对对话的过程打分（value function）。

第一件事就是文本特征提取过程（CNN），此处不再赘述。

第二件事有两种处理思路：

（1）把句子作为动作分解成两个过程，输出量化后的动作，再根据量化值生成一句话；

（2）把字作为一个动作，允许连续输出多个动作。

第三件事最为重要，一个好的评价函数是决定强化学习效率的关键，这里也有两

种思路可以考虑：

（1）以对抗的方式训练一个句子的打分器，但这需要大量的标注的对话语料；

（2）最新的评价一个对话质量的观点是根据对方是否愿意和你聊天来判断，即根据对话的回合数直接对对话打分。

这样就把对话过程建模成一个强化学习的过程了。

综上，对话问题虽然得到了一定的解决，但并没有在所有领域都取得较好的效果，还需要不断优化，目前最好作为辅助功能。另一方面，如果要回答前面提到的常识性的问题，就需要“规则”和“常识”来处理。“规则常识”其实是对实体的一种映射，这种映射需要不停地存入新的常识并更新，所以常识性的问题少不了知识图谱[1]的支持，基于检索式的方法结合知识图谱的使用，将使问题回答的准确性有一个质的提高。

这样就引来了如下思考，如果前面的检索方式并不能够解决多轮对话问题，假设出现这样的一个问题“他是做什么的呢？”，那么我们光看这句话明显不知道“他”指的是谁？此时就需要使用指代消解来解决。笔者所在的团队在这方面已经有了初步的效果，目前已经能实现 5 轮[2]左右的对话。这里简单例举几种指代的处理方式。

（1）原因询问。

A：她昨天又没来。B：为什么？

“为什么”就指前一句话的内容，这时候 B 问的是为什么她昨天又没来。

（2）同义替换。

A：今天天气怎么样。B：今天天气不错。A：明天呢？

明天呢？替换成明天天气怎么样？这里将明天和今天替换，这是同义词性的替换。

（3）结构补充。

---

1 简单来说知识图谱就是从各种结构化非结构化数据种抽取实体、实体间的关系，并构成图。

2 超过 5 轮也可以实现，但为了更像人类，这里仅取 5 轮。

A：周杰伦是干啥的？B：唱歌的。A：出过哪些专辑？

这里“出过哪些专辑？”替换成“周杰伦出过哪些专辑？”这是结构缺失，要去前文找相似结构位置的主题词并补充。

以上三种情况已经可以涵盖多数场景，读者可以根据这些思路解决更多的指代问题，为多轮对话贡献自己的力量。

对话的部分讲到这里可以告一段落了，人类对于智能机器人的渴求已越来越迫切，而智能的重要表征就在于对话能力，对话能反映出智能的程度，这方面的研究也只是刚刚开了个头，还有很多未知的领域需要我们去探索。笔者非常庆幸我们能生活在这个时代，有信心在有生之年看到真正的智能产生。

4 视觉识别

英特在完成了对话机器人项目后，公司新成立了一个业务部门，准备做些视觉相关的项目，主要用在图片的分类和机器人的视觉系统上，并准备下半年将这些项目投放市场试试水。英特调查了市场已有的产品以及技术，准备从人脸识别开始，其中第一个进行的就是人脸表情识别的应用。

表情的识别是一个综合性课题，过程大致是给定一张图片，如果图中有人脸便将其找出，找出之后需要对这张人脸进行分析，判别到底是哪种表情的人脸。其中有两个部分涉及了模型的训练及调优：人脸检测模型和表情识别分类器（如图 4-1 所示）。

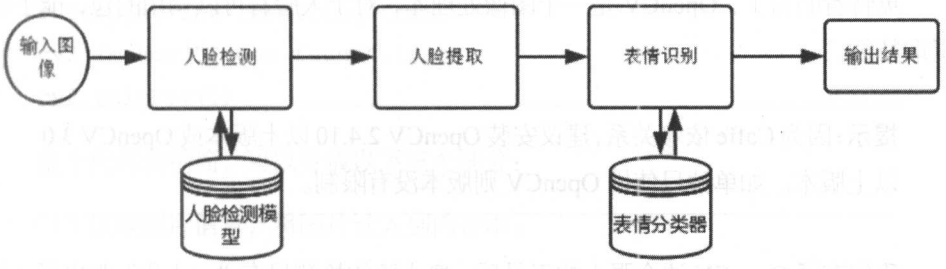


图 4-1　表情识别流程图

4.1 从人脸识别开始

4.1.1 OpenCV 能做什么

英特又开始调研了，要做人脸的表情识别需要先找到图像中人脸的位置，换句话说需要将人脸检测出来。经过一番调查，英特首先选择了 OpenCV 作为人脸检测的基础库。

> OpenCV 是一个用于图像处理、分析、机器视觉方面的开源函数库，它是完全免费的，因此无论是做科学研究，还是商业应用，OpenCV 都可以作为理想的工具库。该库采用 C 及 C++语言编写，可以在 Windows，Linux，mac OSX 系统上运行。由于它更专注于设计成为一种用于实时系统的开源库，它的所有代码都经过优化，计算效率很高。它包含了横跨工业产品检测、医学图像处理、安防、用户界面、摄像头标定、三维成像、机器视觉等领域的超过 500 个接口函数。
>
> 同时，由于计算机视觉与机器学习密不可分，它也包含了比较常用的一些机器学习算法。或许，很多人知道，图像识别、机器视觉在安防领域有所应用。但，很少有人知道，在航拍图片、街道图片（例如 Google street view）中，更需要严重依赖机器视觉的摄像头标定、图像融合等技术都有应用。[1]

英特看明白了，OpenCV 是一个图像处理库，对于人脸有可以调用的包，能手动提取特征。

> **提示：**因为 Caffe 依赖关系，建议安装 OpenCV 2.4.10 以上版本或 OpenCV 3.0 以上版本。如单独只使用 OpenCV 则版本没有限制。

我们有了 OpenCV 这个强大的工具后，离人脸的检测只有“一小步”距离了，下面我们来看一段最简单的人脸检测代码，看看这“一小步”是如何实现的。

1 摘自 OpenCV 官网描述。

```
import cv2
cascPath = "haarcascade_frontalface_alt.xml"
imagePath = "./images-10.jpg"
faceCascade = cv2.CascadeClassifier(cascPath)

# Read the image
image = cv2.imread(imagePath)
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Detect faces in the image
faces = faceCascade.detectMultiScale(
        gray,
        scaleFactor=1.05,
        minNeighbors=3,
    # flags = cv2.cv.CV_HAAR_SCALE_IMAGE
)

print "Found {0} faces!".format(len(faces))

# Draw a rectangle around the faces
for (x, y, w, h) in faces:
    cv2.rectangle(image, (x, y), (x+w, y+h), (0, 255, 0), 2)

# Display the resulting frame
cv2.imshow("Faces found",image)
cv2.waitKey(0)
```

整个代码很精简，可以分成以下三个部分：

（1）读取图片信息，将图片读入到内存中；

（2）加载模型，并调用 faceCascade.detectMultiScale 对图片做检测；

（3）将检测到的位置画在图片上，在上面的代码中是画一个框，然后调用 imshow 输出展现出来。

下面，我们再来具体地分析下代码。

1. 特征分类器

现在将注意力移到代码中的第二句。

```
cascPath = "haarcascade_frontalface_alt.xml"
```

这个 haarcascade_frontalface_alt.xml 是什么？它就是人脸的特征分类器，这些分类器是官方已经训练好的人脸模型，可以直接使用。常用的特征分类器分为 Haar 和 LBP 特征分类器，LBP 的准确率相对于 Haar 要略低些，识别速度却快了很多。英特优先考虑准确率，选用了 haarcascade_frontalface_alt.xml 这个人脸的 Haar 特征分类器，该文件中会描述人脸的 Haar 特征值。当然 Haar 特征的用途可不止于用来描述人脸，也可以用来描述眼睛、嘴唇或者其他物体的具体信息。

现在需要找到 haarcascade_frontalface_alt 并将它加入到项目中，那么在哪里找人脸的 Haar 特征分类器？

OpenCV 有已经自带了人脸的 Haar 特征分类器。OpenCV 安装目录中的\data\haarcascades 目录下的 haarcascade_frontalface_alt.xml 与 haarcascade_frontalface_alt2.xml 都是用来检测人脸的 Haar 分类器。haarcascades 目录下还有人的全身、眼睛和嘴唇的 Haar 分类器。读者可以仿照此例子试验效果。

2. 图片读取

接下来看这两句：

```
image = cv2.imread(imagePath)
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

第一句是将图像文件读取到内存，第二句是将图像变为灰度图，Haar 的特征分类器对于颜色有要求，所以要提前转换好图片的格式。

3. 检测参数

接着往下看：

```
# Detect faces in the image
```

```
faces = faceCascade.detectMultiScale(
        gray,
        scaleFactor=1.05,
        minNeighbors=3,
 minSize=(30, 30)
)
```

（1）minSize、maxSize：检测物体的范围。

修改 minSize（检测的对象最小尺寸，单位：像素×像素）和 maxSize（检测对象的最大尺寸），使对象落在检测器的大小范围内；一般设置 minSize 即可。

（2）minNeighbor：最小邻居数量。

设置 minNeighbor 为 0，观察所有的邻居数量，邻居数量最集中的位置为最终的检测结果。增加邻居数量过滤误检测的对象，检测到对象的条件越苛刻；减少邻居数量增加检测到对象的几率，检测到对象的条件越宽松。

（3）scaleFactor：比例变化因子。

被检测对象的比例变化，比例变化的图像的集合可以构成图像金字塔。scaleFactor 的合理的范围在 1.1—1.4 之间，表明检测人脸时每次按多大的比例来放大，这个比例因子越大，越容易漏掉检测的对象，但检测速度加快；比例因子越小，检测越细致准确，但检测速度变慢。

在以上所有的参数都设置好之后，我们就可以用这段代码来检测图片了。图 4-1 为 OpenCV 使用 Haar 分类器的检测结果。

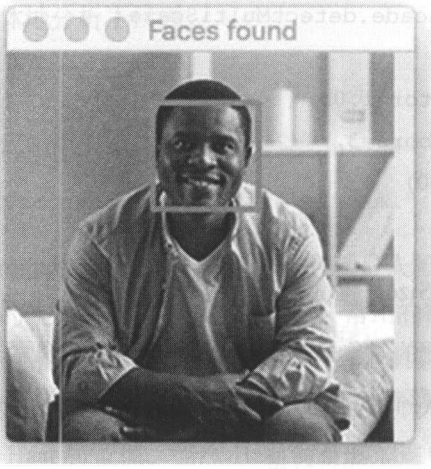


图 4-2　Haar 分类器的检测结果

4. 从图片检测到视频流检测

在大多数场景中我们都需要对视频流进行检测，而不仅仅检测图片。所以英特为这段代码加上了摄像头实时识别功能：

```
cascPath = "haarcascade_frontalface_alt.xml"
faceCascade = cv2.CascadeClassifier(cascPath)

video_capture = cv2.VideoCapture(0)

while True:
    # Capture frame-by-frame
    ret, frame = video_capture.read()

    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    faces = faceCascade.detectMultiScale(
        gray,
        scaleFactor=1.06,
        minNeighbors=8,
    )
```

```
    # Draw a rectangle around the faces
    for (x, y, w, h) in faces:
        cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 0), 2)

    # Display the resulting frame
    cv2.imshow('Video', frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# When everything is done, release the capture
video_capture.release()
cv2.destroyAllWindows()
```

其中 ret, frame = video_capture.read()是用来捕捉摄像头视频流的，另外我们需要保持良好的操作习惯，用完就释放掉：video_capture.release()。

这里要注意的是，由于 Haar 检测方法的性能较低，特别是在一些运算性能较低的机器上（比如树莓派 2），为达到较为流畅的识别效率（这个较为流畅大概在 12 帧左右，几乎是在不改源码的方式下的极限了），需要减少处理数据量，而减小摄像头输出尺寸就是减少数据量最直接的方式。

```
video_capture.set(3, 640)
video_capture.set(4, 480)
```

以上代码将摄像头的捕捉范围改变成 640×480，这样就得到一个用摄像头实时捕捉人脸的代码。

4.1.2 检测精度的进化：Dlib

OpenCV 这段代码用在产品中后，英特团队发现检测人脸时，图片上如果有多张人脸，有部分人脸无法识别出来；另外识别侧脸时识别精度较低，无法达到市场的要求。为解决这个问题，英特再次调研了相关技术，发现一个检测精度可以达到要求的人脸检测库，那就是 Dlib。

*Dlib 是一个跨平台的 C++公共库，具有除了线程支持，网络支持以外，*还提供测试以及大量工具等优点。Dlib 也是一个强大的机器学习的 C++库，包含了许多机器学习常用的算法。Dlib 同时还包含了大量的图形模型算法，其中就包含了人脸识别。

在人脸识别领域 Dlib 实现了 One Millisecond Face Alignment with an Ensemble of Regression Trees[1]中的算法。

人脸检测是一个基础课题，有很多值得研究的地方。英特为了快速达到表情识别的最终目标，并没有在这个节点上多做停留，而直接使用了公开模型，在取得较好的检测效果后就继续其他的研究了。咱们来看一个 Dlib 实现人脸检测的官方例子。

```
import sys
import dlib
from skimage import io

# You can download the required pre-trained face detection model
here:
#
http://dlib.net/files/shape_predictor_68_face_landmarks.dat.bz2
predictor_model = "shape_predictor_68_face_landmarks.dat"

# Take the image file name from the command line
file_name = 'web-1.jpg'

# Create a HOG face detector using the built-in dlib class
face_detector = dlib.get_frontal_face_detector()
face_pose_predictor = dlib.shape_predictor(predictor_model)

win = dlib.image_window()
```

1 http://www.csc.kth.se/~vahidk/papers/KazemiCVPR14.pdf，作者为 Vahid Kazemi 和 Josephine Sullivan。

```
    # Take the image file name from the command line
    # file_name = sys.argv[1]

    # Load the image
    image = io.imread(file_name)

    # Run the HOG face detector on the image data
    detected_faces = face_detector(image, 1)

    print("Found {} faces in the image file
{}".format(len(detected_faces), file_name))

    # Show the desktop window with the image
    win.set_image(image)

    # Loop through each face we found in the image
    for i, face_rect in enumerate(detected_faces):
        # Detected faces are returned as an object with the coordinates
        # of the top, left, right and bottom edges
        print("- Face #{} found at Left: {} Top: {} Right: {} Bottom:
{}".format(i, face_rect.left(), face_rect.top(),
face_rect.right(), face_rect.bottom()))

        # Draw a box around each face we found
        win.add_overlay(face_rect)

        # Get the the face's pose
        pose_landmarks = face_pose_predictor(image, face_rect)

        # Draw the face landmarks on the screen.
        win.add_overlay(pose_landmarks)
```

```
dlib.hit_enter_to_continue()
```

以上这段代码基本上与 OpenCV 代码功能一致，这里就不再解读了。其中英特用 predictor_model = "shape_predictor_68_face_landmarks.dat" 替代了 OpenCV 方法的 haarcascade_frontalface_alt.xml，代码甚至比 OpenCV 还简单，连参数都不用设置就能取得较好的结果。并且如果你需要，还可以将眼睛、鼻子、嘴巴、脸型的特征点标识出来（如图 4-3 所示）。

图 4-3　Dlib 的检测结果

当然如果你要做一个识别人脸的 App 或者业务软件，就要考虑在各种环境下对脸部图像的还原。不同的姿态、光照，有无遮挡、模糊图片处理等都对人脸识别的精度有很大影响。这主要涉及图像变形及处理部分，不在此赘述。

从严格意义上来讲，Dlib 与 OpenCV 的精度及所用的模型有关，OpenCV 也可以提高到 Dlib 差不多的精度。另外 Dlib 包含 CNN 网络，可以直接调用其 API 进行深度学习方面的开发，但有个问题还没有很好的解决，虽然它在训练的时候可以用到 GPU 加速运算，但是在预测时用不了 GPU——这在需要高性能的场景下几乎是致命的。请注意，如果要 Dlib 支持 GPU 运算则需修改框架源码。

4.1.3 表情识别：Openface

英特现在已经将人脸的检测精确度提高到一个可用的范围内，像一些比较偏的面部也能准确检测出来了。

现在我们可以和英特一起进入到表情识别研发阶段，这里我们将表情识别定义为一个多分类问题，针对图片多分类问题的解决方法有很多，最简单的我们可以使用一种 CNN 网络来直接做分类。为了进一步简化问题，我们使用 Openface 库来解决表情识别问题。当然 Openface 远不止提供一个训练器，它还把上述的 Dlib 人脸检测结合进来，成为了专门针对人脸检测和分类的工具。

作为人脸分类器使用的 Openface 可以快速地构建模型。不用一行代码，三步就能搞定人脸检测、表情识别。重新复习一下这个流程图（如图 4-4 所示）。

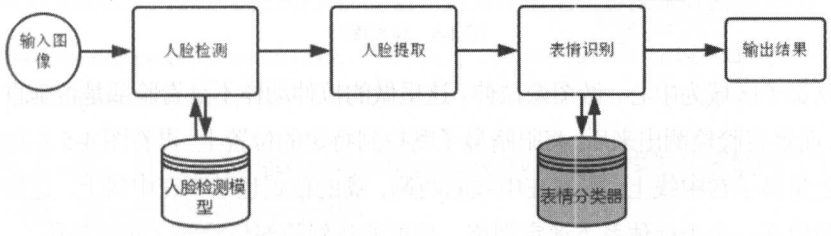


图 4-4 表情识别流程图

此流程图中我们已经学习了人脸检测、人脸提取等步骤，现在的工作是训练一个表情分类器，让机器具有识别表情的能力，我们看看英特是如何训练一个表情分类器的。

第一步 图片预处理

英特先准备了一些简单标记好的人脸图片，一共选取 6+1 类表情（高兴、失望、愤怒、惊讶、悲伤、恐惧、最后一类是正常），每类图片大约 3000 张，尽可能地覆盖各个人种和各个年龄段。图片选取后先进行图片的预处理，主要指姿态的检测和校准。

下面的这行命令用作人脸的检测，当检测完人脸并确认其中有一张人脸时，将继

续做姿态检测和校准。

```
for N in {1..8};do ./util/align-dlib.py ./training-images/ align
outerEyesAndNose ./aligned-images/ --size 96 & done
```

for N in {1..8};do 意思是启动 8 个进程，以最高效率利用 CPU 资源。

我们在表情库的“愤怒”文件夹中选择一张图片，演示姿态检测和校准（如图 4-5 所示）。

图 4-5　拉伸图片

以鼻子区域为中心，做图像拉伸。这里做的拉伸动作不是看脸部是否垂直于水平线上，而是将脸检测出来后，将眼睛鼻子嘴拉到特定的位置上。再看图 4-5 左侧图片，基本上是鼻子在中线上，眼睛在中线的两侧，嘴的位置也基本在中线上。这样将所有的人脸放在一个坐标体系下评定训练，尽可能找到脸部与脸部之间的差异。

第二步　从对齐的图像中生成特征

图片准备好之后，接下来就可以从图中生成特征了，Openface 中有一个预训练好的默认模型[1]：nn4.small2.v1.t7，该模型也是由一个卷积网络训练而成，用作模型预训练。这个卷积神经网络的结构参考了 FaceNet's original nn4 network，并有些改进。图 4-6 为改进网络和各个算法的对比图，虽然没有达到 Deepface（公开最好的人脸算法之一）的识别精度，但也非常不错了，而且在速度上也是平衡得比较好的算法之一。英特使用的测试环境有 Nvidia GTX980ti 和 Nvidia GTX1080，使用 980ti 的单卡训练速度可以用不到 1 小时训练完 5000 张左右的图片。

1　该模型需要提前下载。

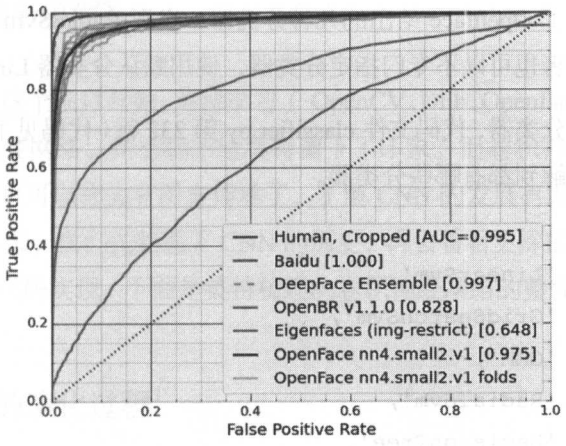


图 4-6 人脸识别算法对比

用该模型再次训练的目的主要是为提取特征工作，默认提取了 128 个点，也就是这个网络的输出层为 128。我们用下面的语句训练经由上一步准备好的脸部图片。

```
./batch-represent/main.lua -outDir ./generated-embeddings/
-data ./aligned-images/
```

运行完成后，目录./generated-embeddings/reps.csv 会包含模型从表情图片中提取的特征向量，每一行特征对应一个图片。

注意：这里的特征生成并不是用人脸检测模型中的 68 个基准点，而是将人脸完整提取后对这部分图片做深度卷积提取特征，进行人脸分类。这 128 个特征点是卷积网络的输出，并不是完全固定不变的位置。

第三步 训练自己的面部检测模型分类器

有了特征点以后，就需要用这些特征点训练分类器，以达到将需要的人脸表情图片区分出来的目的。下面的语句就是用来训练分类器的，分类器输入的就是上一步的脸部特征点，输出就是我们要的最终表情标签了，比如：愤怒、悲伤、快乐等。

```
./demos/classifier.py train ./generated-embeddings --classifier
RadialSvm
```

这里英特对比了 Openface 给出的几种分类器，发现 RadialSvm 对于表情的处理更好一些。当然读者也可以不专门指定分类器，使用默认分类器 LinearSvm。

关于如何指定分类器，代码文件 classifier.py 第 232 行（代码见下）有相关描述，看名字我们可以了解到都有哪些分类器。

```
choices = [
            'LinearSvm',
            'GridSearchSvm',
            'GMM',
            'RadialSvm',
            'DecisionTree',
            'GaussianNB',
            'DBN'],
help = 'The type of classifier to use.',
default = 'LinearSvm'
```

英特用这个模型实时分析摄像头视频流数据，经过了优化后，得到一个可以实时监控人脸表情的系统（如图 4-7 所示）。

图 4-7　表情实时识别结果

4.2 深度卷积网络

从表情识别这个项目开始，英特学习了 OpenCV、Dlib、Openface 特别是 Openface 中使用的卷积神经网络，其准确率给英特留下了印象。而卷积神经网络（CNN，下同）发展到现在也可以称为有历史传承了，了解 CNN 的发展有助于我们更好地理解 CNN 网络。本小节，我们先熟悉下 CNN 的历史，再去看看更深的卷积网络是如何实现的，接着再看代码实现。下面是英特梳理的卷积神经网络的整体演化过程，我们一起来了解下。

4.2.1 CNN 的演化过程[1]

先来回顾下 CNN 的结构演化历史（如图 4-8 所示）。

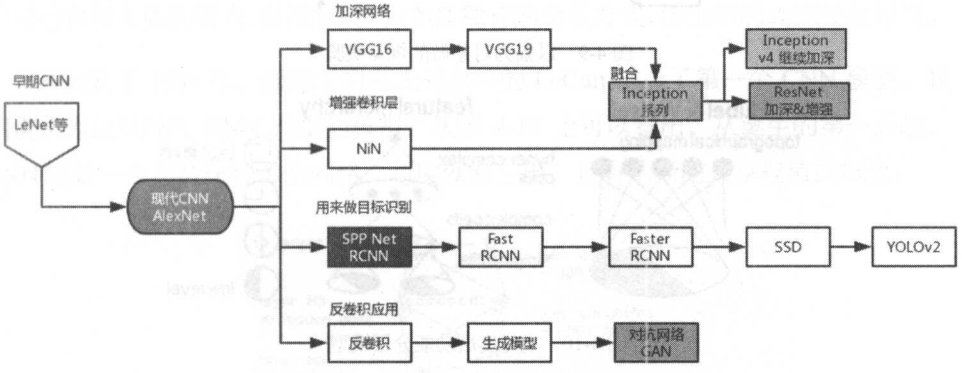


图 4-8 CNN 演化图

CNN 的起点是神经认知机模型，虽然 1980 年神经网络就开始出现了卷积结构，但第一个 CNN 模型的诞生已经是 1989 年，直到 1998 年才诞生了 LeNet。随着 ReLU 和 dropout 的提出，以及 GPU 并行计算和大数据爆发带来的红利（大量的图像数据可用来训练），CNN 在 2012 年迎来了历史突破。2012 年之后，CNN 的演化路径可以归结为四条：（1）更深的网络，（2）增强卷积的功能，（3）目标检测，（4）反卷积。下面我们详细介绍一下 CNN 的演化过程。

1 本节部分内容改编自《CNN 的近期进展与实用技巧》，刘昕，https://zhuanlan.zhihu.com/p/21432547。

1. CNN 起源

CNN 的起源仍然要追溯到仿生学的成功，1962 年 Hubel 和 Wiesel 用猫做了实验[1]，提出了视觉皮层的功能模型，从简单细胞到复杂细胞再到超复杂细胞（如图 4-9 和图 4-10 所示）。

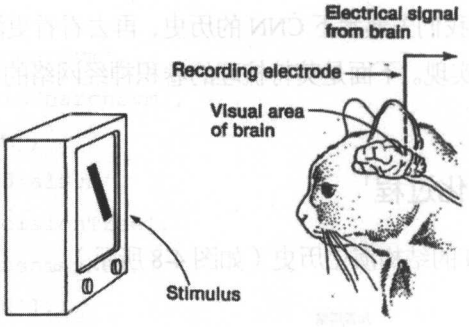


图 4-9　以猫为对象做的视觉实验

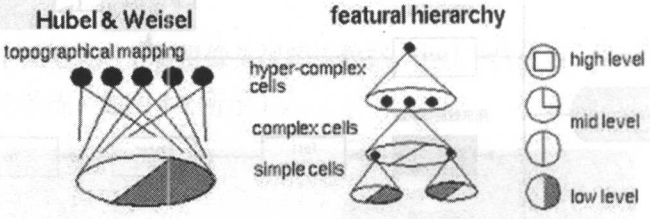


图 4-10　视觉皮层功能分层模型

受此启发，1980 年，KUNIHIKO FUKUSHIMA 提出，将简单细胞实现为卷积，复杂细胞实现为池化层。神经认知机采用自组织的方式进行无监督的卷积核训练，由于训练方式上和现有的卷积网络有很大差别，因此这并不是现代通过后向传播算法端到端训练的 CNN（如图 4-11 所示）。

1　现在你也可以在 YouTube 上找到相应的视频 https://www.youtube.com/watch?v= 8VdFf3egwfg。

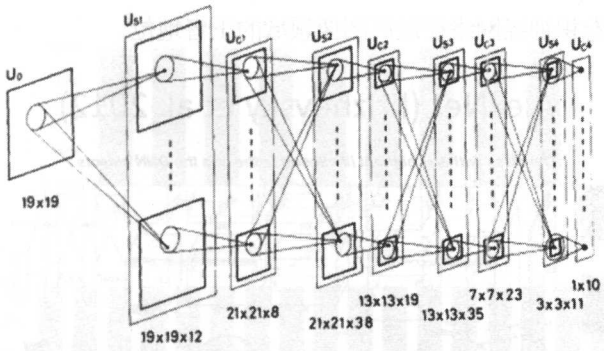


FIGURE 2. Hierarchical network structure of the neocognitron. The numerals at the bottom of the figure show the total numbers of S- and C-cells in individual layers of the network which are used for the handwritten numeral recognition system discussed in Section 4.

图 4-11　基于分层网络的神经网络结构

这个网络中还是以 S-细胞和 C-cell 细胞称呼独立层的功能，这两个称呼都来自于对动物和人体的研究，并没有后期 CNN 独有的命名方式，仿生学的成就处处可见。

时间到了 1989 年，深度学习三大神之一的 LeCun 创造了第一个 CNN 模型。我们也认为这是现代 CNN 模型的鼻祖，从图 4-12 上可以看出，从诞生的第一天起，CNN 就是一个层数比较多的网络，隐层就有三层，且有清晰的共享权值的概念。

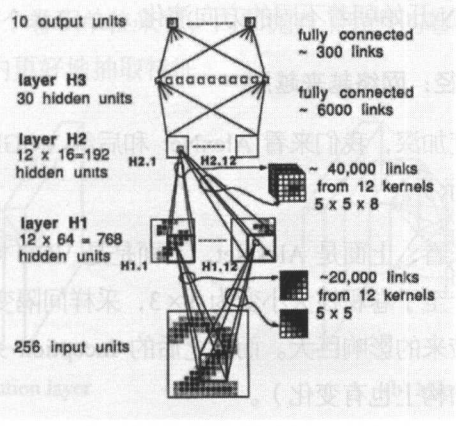


图 4-12　最早的 CNN 模型

只不过这个网络的准确率，仍然比当时以人工提取特征方式为代表的模式识别方法要差。转折在 2012 年到来，Krizhevsky 携 AlexNet（如图 4-13 所示）在当年的 ImageNet 图像分类竞赛中，以 top-5 的错误率、比上一年冠军下降了十个百分点的绝

对优势，宣告了卷积神经网络在以后图像领域的王者地位。

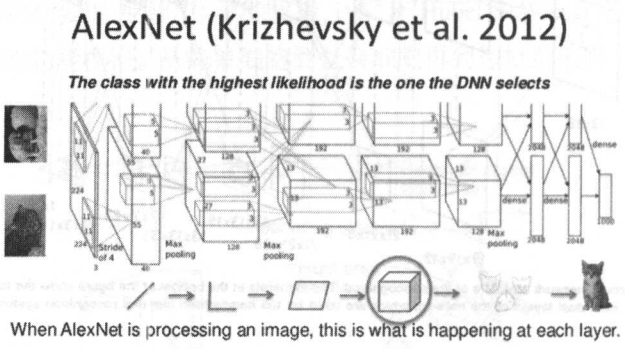


图 4-13　AlexNet 模型

AlexNet 的成功不只是在 LeCun 网络上小修小补，Alex 中提出了两个新观点：一是 ReLU 模拟人类神经元，二是为防止过拟合提出的 Dropout[1]。但是，有了好的网络而没有训练集，CNN 仍然得不到大规模推广，好在 2012 年 Hadoop 随着大数据的应用开始推广普及，同时，GPU 作为并行训练的加速器，也极大地推动了 CNN 的发展。

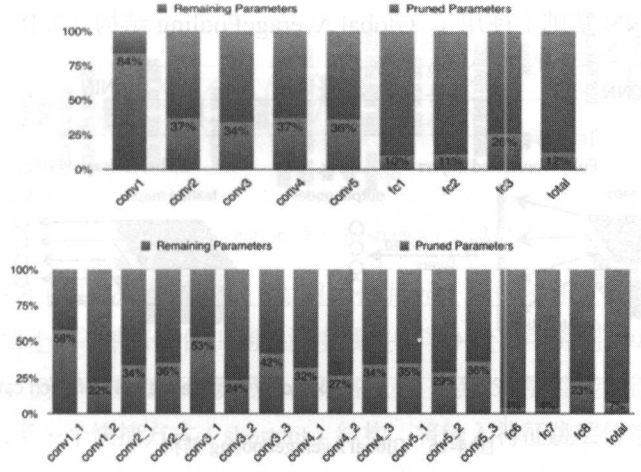
2012 年之后，CNN 开始朝着不同的方向演化。

2．第一条演化路径：网络越来越深

为了解网络的深度加深，我们来看 AlexNet 和后继 VGGNet 的横向网络结构层数对比（如图 4-14 所示）。

我们将网络横过来看，上面是 AlexNet，下面是是 VGG 网络，直观地就可以对比出层数差异的巨大。至于卷积核大小变为 3×3，采样间隔变为 1×1，及池化层的变化都不如层数加深带来的影响巨大。而这之后的 Inception 系列结构，都是在深度上越来越深（当然在结构上也有变化）。

1　卷积神经网络的过拟合很常见。

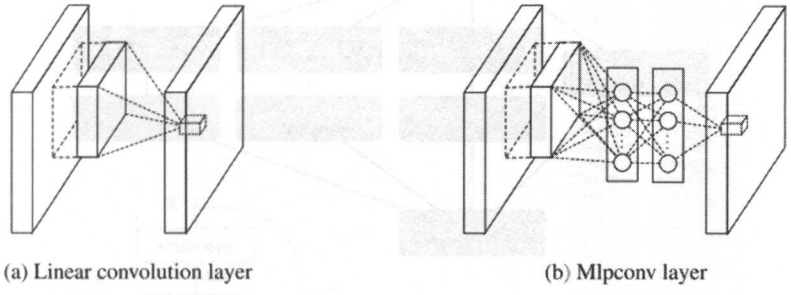


et al. Learning both Weights and Connections for Efficient Neural Networks, NIPS 2015

图 4-14　AlexNet 和 后继 VGGNet 层数对比[1]

3. 第二条演化路径：卷积模块的内部变化

2013 年底，颜水成提出了 NIN（Network in Network）结构（如图 4-15 所示），并于 2014 年取得了 ILSVRC 物体检测竞赛的冠军。之后，增强卷积的概念深入人心。在 NIN 网络中，取一个卷积单位来看，内部抛弃了线性的卷积方式，而加入了多层隐层，这样能在单元内更好地抽取特征。



(a) Linear convolution layer　　(b) Mlpconv layer

图 4-15　NIN（Network in Network）结构

1　引自 Han et al. *Learning both Weights and Connections for Efficient Neural Networks*, NIPS 2015。

另外在原有 CNN 基础上使用了 Global AveragePooling 结构（如图 4-16 所示）。

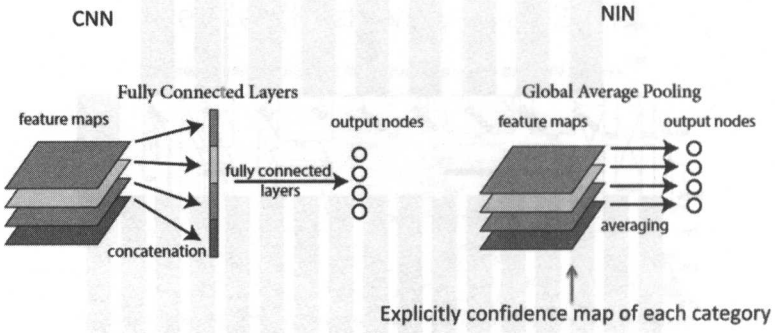


图 4-16　Global AveragePooling 结构

Google 在 NIN 基础上于 2014 年提出了 GoogLeNet（Inception v1），并随后改进出 Inception v2、v3 和 v4。

Inception 结构（如图 4-17 和图 4-18 所示）也是一种增强的卷积结构，这次的变换将模块内部增加多个卷积层用并联或串联的方式组合在一起，用于替代之前的单一卷积层。目标也是为了更有效率地提取特征。

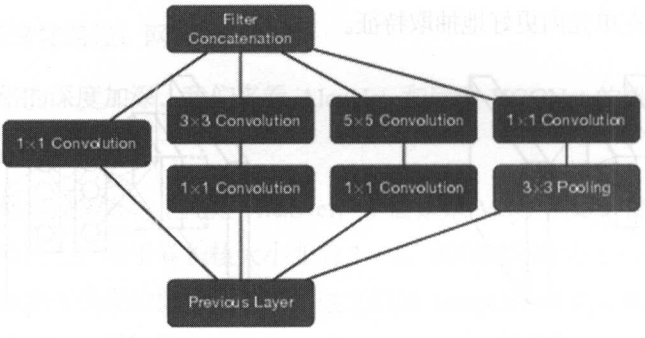


图 4-17　Inception 内部小网络结构

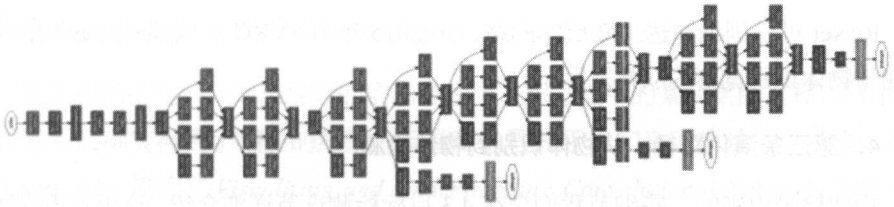


图 4-18 Inception v1（GoogleNet）完整结构

Residual Net（深度残差网络）（如图 4-19 所示）可以看作是前两条演化路径的集成。何凯明博士在《基于图像识别的深度残差学习》论文中提到，单纯增加深度会导致网络退化，例如 CIFAR-10 数据集，网络从 20 层到 56 层性能反而下降了。为此 ResNet 中引入了一个快捷方式（shortcut）结构，将输入特征跳层传递与卷积的结果相加。

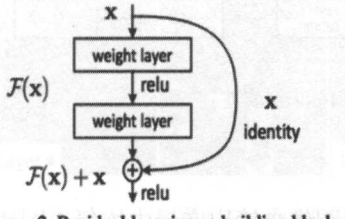


Figure 2. Residual learning: a building block.

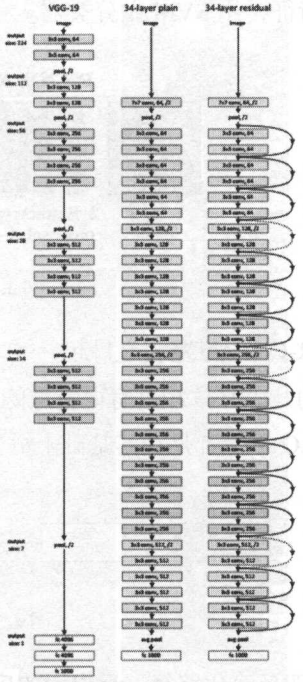


Figure 3. Example network architectures for ImageNet. **Left**: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. **Middle**: a plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. **Table 1** shows more details and other variants.

图 4-19 ReNet 网络结构

ReNet 可以训练深达 152 层的网络，是 2015 年 ILSVRC 不依赖外部数据的物体检测与物体识别竞赛的双料冠军。

4．第三条演化路径：从物体识别到物体检测

物体检测识别的一些细节我们会在 4.3 目标检测这节详细介绍，这里着重描述其在历史中扮演的角色。

R-CNN（如图 4-20 所示）的思想源自一个很简单的目的，卷积神经网络在图像的特征提取上已经应用得非常成功了，是否可以用卷积网络来做物体识别呢？2013 年 11 月，Ross Girshick 等发表了一篇论文，内容就是利用卷积网络来做物体识别。其核心思想就是将物体识别问题转化成一个图像分类问题，这样就能让 CNN 无缝接手了。R-CNN 用最简单的语句来描述就是：用 selective Search 做物体的抽取，用 CNN 提取图像特征，用 SVM 做分类器。

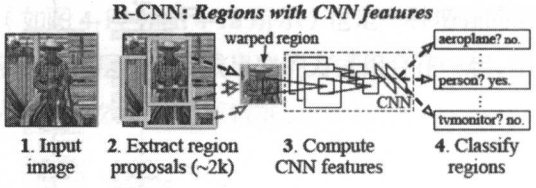


图 4-20　R-CNN 结构

然而 R-CNN 自发明之日起，效率就低得令人发指，但这篇文章作为用 CNN 做目标识别的开山文章却具有很特殊的地位，在此之后的 Fast-RCNN、Faster-RCNN、SSD、YOLOv2 等都是在其基础上衍生改进而来的（如图 4-21 所示）。

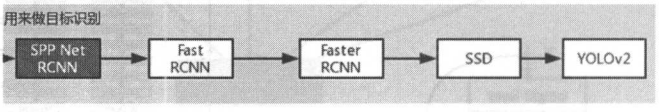


图 4-21　目标识别演进

5．第四条演化路径：反卷积的应用

反卷积（deconvolution）是指从已知图像中输出向量中恢复出图片原有的像素，也就是将卷积层输入和输出颠倒过来，这也是卷积过程的逆操作，可以叫它反卷积网络。当然反卷积不可能只是这么简单地将特征映射成像素，还有很重要的概念比如反

池化层（unpooling），但基本原理大致如此，这里不做过多探讨。

反卷积网络在一开始出现时的应用是还原卷积过程中的某一层的过程，人们因为使用卷积后确实得到了相应的结果，但卷积过程对于人们来说就是一个黑盒。2015年CVPR的一篇论文 *Visualizing and Understanding Convolutional Networks* 将卷积过程可视化了，才让人们很好地观察到了每一层卷积到底做了什么事（如图4-22所示）。

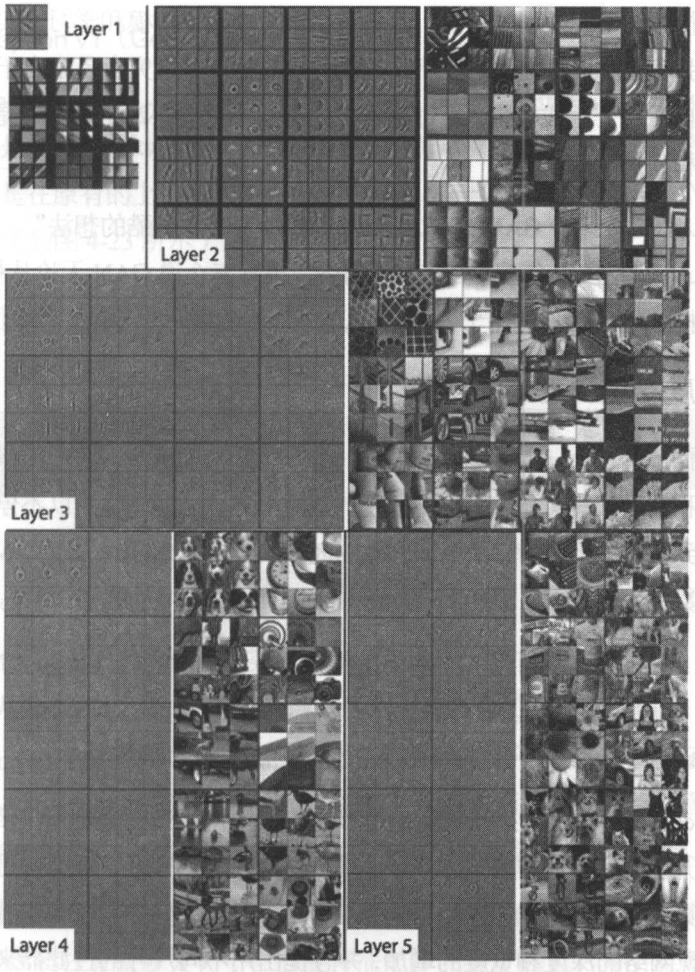


图4-22 反卷积的应用：可视化[1]

1 引用自论文 *Visualizing and Understanding Convolutional Networks*。

*你看到了可能会说，反卷积只是能干这个用的啊？有必要单独列一条说明吗？嗯，*2015 年之前的情况都是这样的，但在生成式对抗网络（GAN）爆发后，以 GAN 现在如日中天的地位，反卷积作为其中不可分割的基础部分就有必要单独列出了。

那么生成式对抗网络（GAN）是什么？

GAN 建立了一个学习框架，实际上就是模拟两方：生成模型和判别模型，并让它们不停地拆台。生成模型的目的，就是要尽量去模仿、建模和学习真实数据的分布规律；而判别模型则是要判别自己所得到的一个输入数据，究竟是来自于真实的数据分布还是来自于一个生成模型。通过这两个内部模型之间不断的竞争，提高两个模型的生成能力和判别能力。

Yann LeCun 曾评价 GAN 是“20 年来机器学习领域最酷的想法”，这里有必要提一句 LeCun 是在 1989 年提出的 CNN 网络。不管怎么说 GAN 无论从机制到实现，都是经得起推敲的，并且适用于各种领域，这也必然奠定了它的王者地位。

4.2.2 深度卷积和更深的卷积

前面我们说到从 AlexNet 结构之后第一条演化路径就是网络的层数更深，进入了一个深度卷积的时代。人类对于识别精度的追求是永无止境的，所以不停地加深网络的层数以达到更好的准确率。从 AlexNet 的 8 层到 VGG16 的 16 层，到 VGG19 的 19 层，网络层数一直在增加，从 Inception v1（Googlenet）增加到了 22 层。

如果把 Inception 系列之前的网络结构称为深度卷积网络（deep CNN），那么 Inception v1 之后的网络应该被称为更深的卷积网络（deeper CNN），从这里开始卷积网络层数已经刹不住车了，直到在残差网络中出现 152 层结构。

那么 Inception 是什么？Inception 是 Google 提出的一个系列的网络结构，其命名也有开端或开创之意，可以看出 Google 对此模型的重视程度。该结构可以充分利用网络中“计算资源”（充分开发和利用每层提取的特征），在保证固定计算复杂度的前提下，实现网络的深度和宽度的增加，并且提出用不同尺寸的过滤器来处理不同尺寸的图片。

其架构的一个重点是 network-in-network，顾名思义是在神经网络两层中又嵌入

一个小网络，以取得性能和准确性的平衡。而 Inception 大量应用了这种类型的网络结构（network-in-network），让我们更方便地增加深度和增加宽度。

Inception 网络可以帮助我们了解深度卷积网络中的主要设计思想，而了解这些思想有助于我们设计出精度更高、更为复杂的网络。

目前 Inception 系列已经进化到第 4 代，下面我们一起来看看进化过程。

1. Inception v1（GoogLeNet）模型

Inception v1 的网络，也被称为 GoogLeNet，前文已经提到，Inception v1 结构的一个重点就是引入了 network-in-network，简称 NIN。什么是网络中的网络呢（NIN）？简单来说即是在原有的上下两层网络结构中加入了一个几层卷积，和输入输出层构成一个子网络（如图 4-23 所示）。

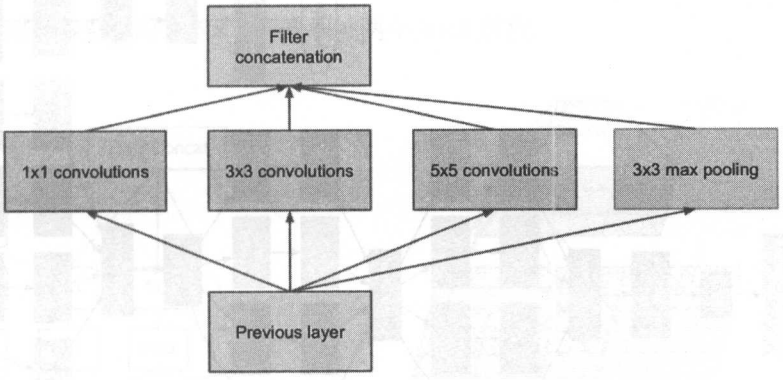


图 4-23　Inception v1 的原始版本结构

Inception v1 的原始版本结构（图 4-23）将 1×1，3×3，5×5 的 conv 和 3×3 的池化，堆叠在一起，一方面增加了网络的宽度，另一方面增加了网络对图片尺寸的适应性；这和上下两层一起构成了一个子网络结构，这就是 NIN 的思想。

但该原始结构有个问题，所有的卷积核都在上一层的所有输出上来做，那 5×5 的卷积核所需的计算量就太大了，造成特征矩阵的维度很大。为了避免这一现象，Inception 提出，在 3×3 前，5×5 前，Max Pooling 后分别加上了 1×1 的卷积核，起到降低特征矩阵维度的作用，也起到增加非线性的作用（如图 4-24 所示）。

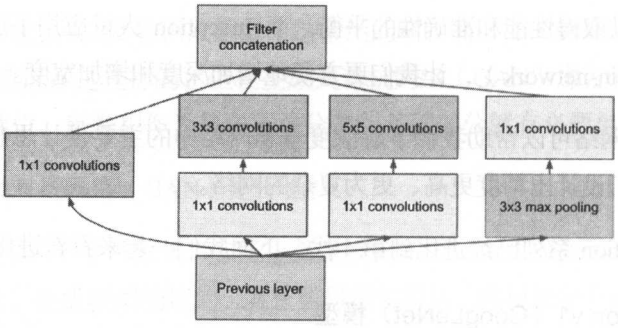


图 4-24　Inception v1 的改进版本结构

v1 的另一个改进点是深度加深，图 4-25 为前文已经提到的 Inception v1（GoogLeNet）的部分结构图。

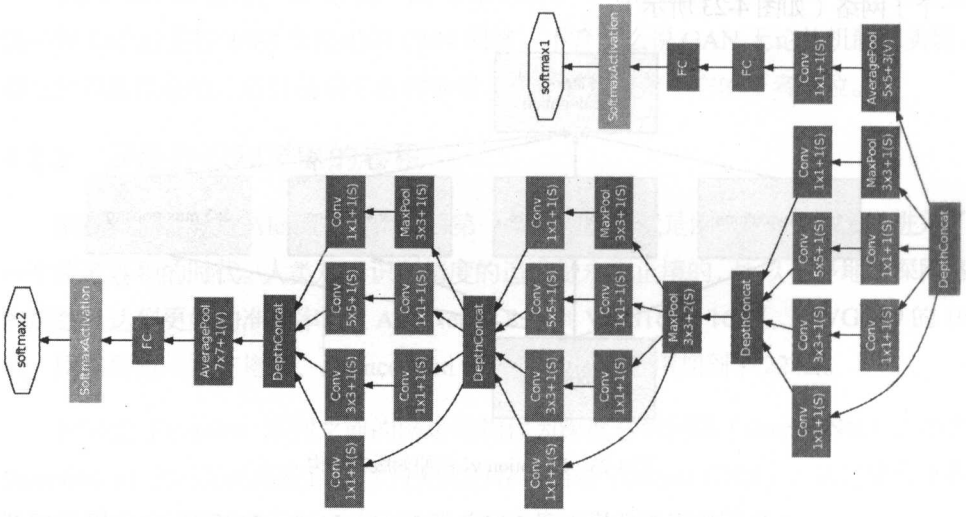


图 4-25　Inception v1（GoogLeNet）部分结构图

2. Inception v2 模型

v1 论文发表之后，Inception 作者继续优化其结构，在 v2 版本中一方面加入了 BN 层，减少了内部神经元的数据分布变化，使每一层的输出都规范化到一个 N(0,1) 的高斯分布；另外一方面学习 VGG 网络，用两个 3×3 的 conv 替代 Inception 模块中的 5×5，既降低了参数数量，也加速了计算。图 4-25 列出 Inception v1 的改进版本结构，读者可以做个对比。

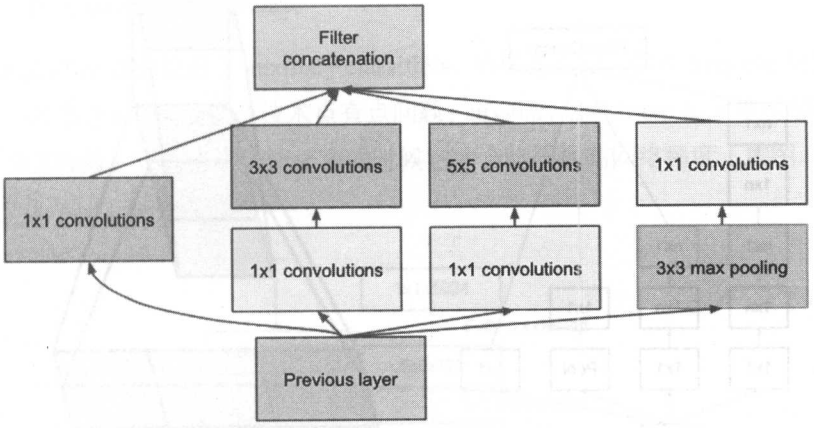
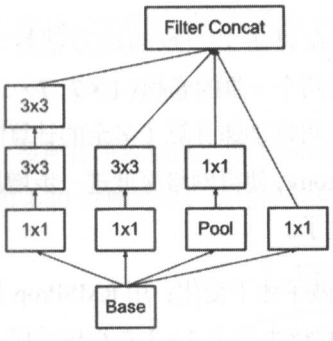


图 4-26　Inception v1 的改进版本结构

图 4-27 左为 v2 对结构的改进，图 4-27 右是改进结构的两个 3×3 卷积示意图，可以清晰地观察到，原有 5×5 的卷积被两个 3×3 替代。



Inception v2 的改进结构

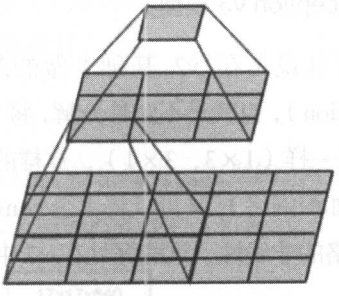
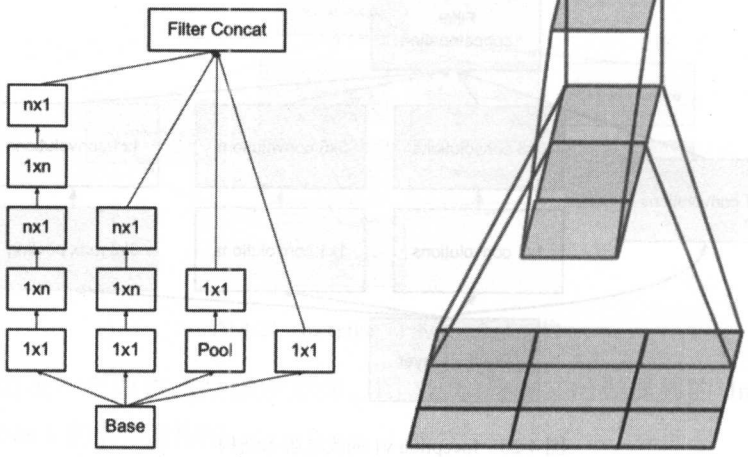


Figure 1. Mini-network replacing the 5 × 5 convolutions.

Inception v2 改进结构示意图

图 4-27　Inception v2 的改进结构

使用 3×3 的已经很小了，那么更小的 2×2 呢？2×2 虽然能使得参数进一步降低，但是不如另一种方式更加有效，那就是非对称（Asymmetric）卷积，什么叫不对称，就是再次使用 1×3 和 3×1 两种来代替原有的 3×3 的卷积核。这种结构在前几层效果不太好，但对层数为 12～20 层的中间层效果明显（如图 4-28 所示）。



非对称卷积结构图　　非对称卷积示意图（3×3 变为 3×1 和 1×3）

图 4-28　非对称卷积

3．Inception v3 模型[1]

v3 可以说是在 v2 基础上做的改进，它们在思想上一致的地方就是分解（Factorization），也就是不对称分解，将 7×7 分解成两个一维的卷积（1×7，7×1），3×3 也是一样（1×3，3×1），这样的好处是，既可以加速计算（多余的计算能力可以用来加深网络），又可以将 1 个 conv 拆成 2 个 conv，使得网络深度进一步增加，增加了网络的非线性，这些好处在 v2 中已经提到过了。

除了与 v2 一致的地方，v3 在原始 v2 的结构上做了如下变化：用 RMSProp 替代 SGD，在类别全连接层后加入 LSR 层，将 7×7 卷积核由三个 3×3 卷积核替代。当然 v3 结构的贡献不至于此，下面我们稍微深入下，看看 v3 结构的其他四个贡献。

（1）Filter 分解。

其实大 filters 拆解成若干小 filters 叠加的方法已经在 VGG 里提过了，Inception v3 更进了一步，提出了“非对称卷积”，简单说就是在卷积模块内，做多种尺度的 filters，替代统一规格的 filters。

1　本小节部分内容引用自 http://www.jianshu.com/p/0cc42b8e6d25，作者 Traphix，稍有改动。

（2）优化 auxiliary classifiers。

GoogLeNet 首次提出了 auxiliary classifiers，效果还不错。作者 Szegedy 同志在论文发布一年多之后，发现这个技术点有点问题：auxiliary classifiers 在训练初期的时候并不能加速收敛，只有当训练快结束的时候它才会略微提高网络精度（如图 4-29 所示）。

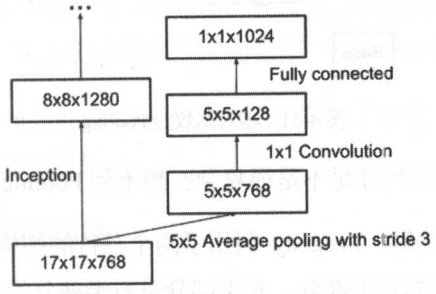


图 4-29　优化 auxiliary classifiers

然后 Szegedy 就把第一个辅助分类器去掉了，理由如下：如果侧分支是批量归一化的或具有丢弃层，则网络的主分类器执行得更好。

（3）新的 Pooling 层

按照传统的做法，在 Pooling 之前，为了防止信息丢失，应当加入了扩展层，如下图 4-30 右侧部分，请注意这层的信息比上一层信息要多。

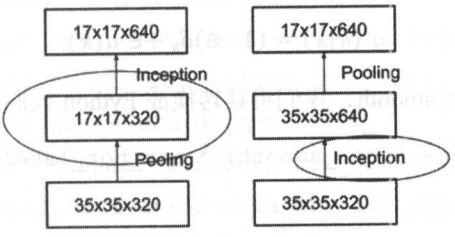


图 4-30　低效的 Pooling 层

这么做有个问题，会增加运算量，于是 Szegedy 就想出了下面这种 Pooling 层（如图 4-31 所示）。

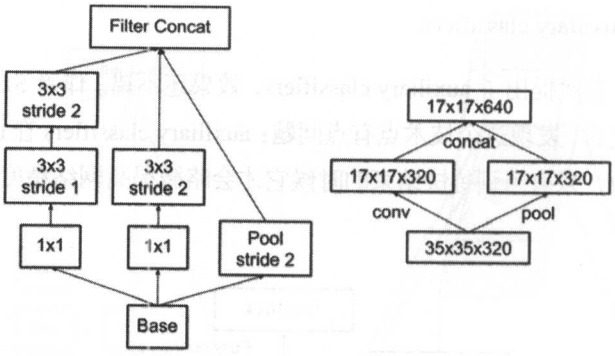


图 4-31　相对高效的 Pooling

更有效率地缩小网络尺寸是不是很复杂？想不到 Pooling 也能玩儿出这么多花样。

图 4-30 可以这么理解，Szegedy 利用了两个并行的结构减少网络尺寸，分别是 conv 和 pool，就是上图的右半部分。左半部分是右半部分的内部结构。

（4）标签平滑（label smooth）。

深度学习用的 labels 一般都是 One-Hot 向量，用来指示 classifier 的唯一结果，这样的 labels 有点类似信号与系统里的脉冲函数，即只在某一位置取 1，其他位置都是 0。

Labels 的脉冲性质会引发两个不良后果：一是过拟合，另一个是降低了网络的适应性。标签平滑就是为了解决这个问题，那么具体是怎么实现的？就是通过下面的公式。

$$q'(k|x) = (1-\epsilon)\delta_{k,}+\epsilon\, u(k)$$

为了方便理解 label smooth，我们将其转换成 Python 代码，如下。

```
new_label= (1.0 - label_smooth) * one_hot_label + label_smooth /
num_classes
```

这个实现很好理解，将为 1 的信号做了消减，对 0 的信号做了补足，这可以让网络不那么容易过拟合了。

v3 的网络实现取值是令 label_smooth = 0.1，num_classes = 1000。Label smooth 使整体网络精度提高了 0.2%。

4. Inception v4 模型

Google 在研究微软发表的 ResNet 的结构时发现该结构可以极大地加速训练，同时性能也有提升，而且避免了深度加深导致精度反而下降的问题，Google 研究人员得到一个 Inception-ResNet v2 网络，同时为了跟微软对标，还在原有的 Inception 模块基础上设计了一个更深更优化的 Inception v4 模型，能达到与 Inception-ResNet v2 相媲美的精度，但性能始终要差些。Google 研究人员得出结论：Residual Connections 好像只能加速网络收敛，真正提高网络精度还需要“更大的网络规模”。

Inception v4 相比 Inception v3 改进的地方是：增加了模型深度，同时增加了模型宽度，但并没有加入残差的连接。

4.2.3 实现更深的卷积网络

从 Inception v1—v4 中，英特看出了 Inception 系列一脉相承的思想。英特也想试试从头实现一个 CNN 网络。下面就以 Inception v4 为例，使用深度学习框架 MXNet 实现代码[1]。

首先构造一个卷积层函数 Conv，它的卷积核默认为 1×1，stride 为 1×1。stride 指的是卷积核每次移动的间隔，pad 指边缘预留多少个单位不做卷积。

```
    def Conv(data, num_filter, kernel=(1, 1), stride=(1, 1), pad=(0,
0), name=None, suffix=''):
        conv = mx.symbol.Convolution(data=data,
num_filter=num_filter, kernel=kernel, stride=stride, pad=pad,
no_bias=True, name='%s%s_conv2d' %(name, suffix))
        bn = mx.symbol.BatchNorm(data=conv,
name='%s%s_batchnorm' %(name, suffix), fix_gamma=True)
        act = mx.symbol.Activation(data=bn, act_type='relu',
name='%s%s_relu' %(name, suffix))

        return act
```

1 MXNet 目前还没有 v4 算法，这里完全是参考论文编写的代码，以说明卷积网络的结构。

卷积层主体完成后，需要给定激活函数，这里用 relu 作为激活函数：act_type='relu'。

对于卷积核的定义在所有的深度学习框架中看起来是比较统一的，比如在 Keras 框架实现 VGG16 里的一段：

```
model.add(Convolution2D(256, 3, 3, activation='relu'))
```

这句的 256 是定义卷积核的个数，可以认为是卷积层输出的维度，“3, 3”定义卷积核的大小 3×3，接着指明激活函数 activation='relu'。这里插入 Keras 代码目的是希望大家可以举一反三，不必拘泥用一种深度学习框架实现算法。

接着，构造一个 InceptionA 函数对应论文上的 InceptionA 结构。

```
def InceptionA(input, name=None):
    p1 = mx.symbol.Pooling(input, kernel=(3, 3), pad=(1, 1),
pool_type='avg', name='%s_avgpool_1' %name)
    c1 = Conv(p1, 96, kernel=(1, 1), pad=(0, 0),
name='%s_conv1_1*1' %name)

    c2 = Conv(input, 96, kernel=(1, 1), pad=(0, 0),
name='%s_conv2_1*1' %name)

    c3 = Conv(input, 64, kernel=(1, 1), pad=(0, 0),
name='%s_conv3_1*1' %name)
    c3 = Conv(c3, 96, kernel=(3, 3), pad=(1, 1),
name='%s_conv4_3*3' %name)

    c4 = Conv(input, 64, kernel=(1, 1), pad=(0, 0),
name='%s_conv5_1*1' % name)
    c4 = Conv(c4, 96, kernel=(3, 3), pad=(1, 1),
name='%s_conv6_3*3' % name)
    c4 = Conv(c4, 96, kernel=(3, 3), pad=(1, 1),
name='%s_conv7_3*3' %name)
```

```
    concat = mx.symbol.Concat(*[c1, c2, c3, c4],
name='%s_concat_1' %name)

    return concat
```

构造一个 ReductionA 结构，该结构是为了将池化层和卷积层进行合并输出。

```
def ReductionA(input, name=None):
    p1 = mx.symbol.Pooling(input, kernel=(3, 3), stride=(2, 2),
pool_type='max', name='%s_maxpool_1' %name)

    c2 = Conv(input, 384, kernel=(3, 3), stride=(2, 2),
name='%s_conv1_3*3' %name)

    c3 = Conv(input, 192, kernel=(1, 1), pad=(0, 0),
name='%s_conv2_1*1' %name)
    c3 = Conv(c3, 224, kernel=(3, 3), pad=(1, 1),
name='%s_conv3_3*3' %name)
    c3 = Conv(c3, 256, kernel=(3, 3), stride=(2, 2), pad=(0, 0),
name='%s_conv4_3*3' %name)

    concat = mx.symbol.Concat(*[p1, c2, c3],
name='%s_concat_1' %name)

    return concat
```

接着实现 InceptionB、ReductionB 结构，同样也对应论文上 InceptionB、ReductionB 结构。

```
def InceptionB(input, name=None):
    p1 = mx.symbol.Pooling(input, kernel=(3, 3), pad=(1, 1),
pool_type='avg', name='%s_avgpool_1' %name)
    c1 = Conv(p1, 128, kernel=(1, 1), pad=(0, 0),
name='%s_conv1_1*1' %name)
```

```
        c2 = Conv(input, 384, kernel=(1, 1), pad=(0, 0),
name='%s_conv2_1*1' %name)

        c3 = Conv(input, 192, kernel=(1, 1), pad=(0, 0),
name='%s_conv3_1*1' %name)
        c3 = Conv(c3, 224, kernel=(1, 7), pad=(0, 3),
name='%s_conv4_1*7' %name)
        #paper wrong
        c3 = Conv(c3, 256, kernel=(7, 1), pad=(3, 0),
name='%s_conv5_1*7' %name)

        c4 = Conv(input, 192, kernel=(1, 1), pad=(0, 0),
name='%s_conv6_1*1' %name)
        c4 = Conv(c4, 192, kernel=(1, 7), pad=(0, 3),
name='%s_conv7_1*7' %name)
        c4 = Conv(c4, 224, kernel=(7, 1), pad=(3, 0),
name='%s_conv8_7*1' %name)
        c4 = Conv(c4, 224, kernel=(1, 7), pad=(0, 3),
name='%s_conv9_1*7' %name)
        c4 = Conv(c4, 256, kernel=(7, 1), pad=(3, 0),
name='%s_conv10_7*1' %name)

        concat = mx.sym.Concat(*[c1, c2, c3, c4],
name='%s_concat_1' %name)

        return concat

    def ReductionB(input,name=None):
        p1 = mx.symbol.Pooling(input, kernel=(3, 3), stride=(2, 2),
pool_type='max', name='%s_maxpool_1' %name)

        c2 = Conv(input, 192, kernel=(1, 1), pad=(0, 0),
name='%s_conv1_1*1' %name)
```

```
        c2 = Conv(c2, 192, kernel=(3, 3), stride=(2, 2),
name='%s_conv2_3*3' %name)

        c3 = Conv(input, 256, kernel=(1, 1), pad=(0, 0),
name='%s_conv3_1*1' %name)
        c3 = Conv(c3, 256, kernel=(1, 7), pad=(0, 3),
name='%s_conv4_1*7' %name)
        c3 = Conv(c3, 320, kernel=(7, 1), pad=(3, 0),
name='%s_conv5_7*1' %name)
        c3 = Conv(c3, 320, kernel=(3, 3), stride=(2, 2),
name='%s_conv6_3*3' %name)

        concat = mx.symbol.Concat(*[p1, c2, c3],
name='%s_concat_1' %name)

        return concat
```

后面的InceptionC 结构代码就不列出了,感兴趣的同学可以按照论文自己编写。

最后我们将这些子网络结构组合起来。

```
def get_symbol(num_classes=1000):
    data = mx.symbol.Variable(name="data")
    x = Inception_stem(data, name='in_stem')

    for i in range(4):
        x = InceptionA(x, name='in%dA' %(i+1))

    #Reduction A
    x = ReductionA(x, name='re1A')

    for i in range(7):
        x = InceptionB(x, name='in%dB' %(i+1))

    #ReductionB
```

```
        x = ReductionB(x, name='re1B')

        for i in range(3):
            x = InceptionC(x, name='in%dC' %(i+1))

        #Average Pooling
        x = mx.symbol.Pooling(x, kernel=(8, 8), pad=(1, 1),
pool_type='avg', name='global_avgpool')

        #Dropout
        x = mx.symbol.Dropout(x, p=0.2)

        flatten = mx.symbol.Flatten(x, name='flatten')
        fc1 = mx.symbol.FullyConnected(flatten,
num_hidden=num_classes, name='fc1')
        softmax = mx.symbol.SoftmaxOutput(fc1, name='softmax')

        return softmax
```

将 Reduction Inception 构建的模块组合起来后，我们构建一层全局平均 Pooling 层，并紧接着链接 Dropout 层[1]，设定它保留 0.2 也就是 20%的神经元不起作用，而在每次迭代时随机地让剩下的 80%神经元参与工作。这样的好处是能让模型适应更多的图像，专业的说法就是防止过拟合。接着将维度拉成一维向量，flatten = mx.symbol.Flatten(x, name='flatten')，这功能有点像将一个立方体拍平，拍成正方形，所以这里的函数名用的是 Flatten，接着最后一层输出类别，并于上一层全链接。

以上是就是英特用 MXNet 实现的一个 Inception v4 结构。

4.2.4 残差网络的实现

在 CNN 演化历史中我们谈到：CNN 想要提高精度必须朝加深网络深度发展，那么是否能够简单地通过增加更多的网络层次学习到更好的网络？研究者发现随着网

1 dropout 更是一种机制而不是“层”，这里是为了行文方便。

络深度的增加，准确率(Accuracy)增长的速率会很快达到饱和，然后就很快下降了，而且，对一个合适的深度网络模型增加更多的层次，会使得训练误差更高，并且这也早已被实验充分验证了，图 4-32 展示了一个典型的案例。

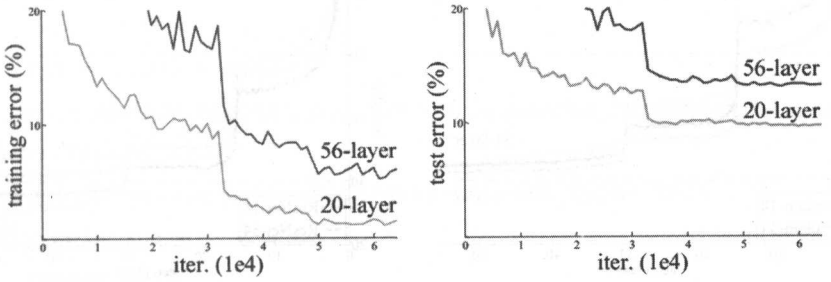


图 4-32　56 层网络比 20 层网络训练、测试误差更高

从中可以看出，深度太深不可避免地会带来网络的退化问题，对训练准确度的退化问题表明了并不是所有的系统都能容易地达到优化。这样就造成了单纯的深度增加网络的准确度还不如 VGG 这样深度的网络。

单纯的深度增加网络还带来了另一个问题，就是随着网络增加，性能逐步降低。这就有必要找到一种高性能的网络结构，以应对网络增加带来的问题，这时候研究者们及时地拿出了残差学习，见图 4-33。

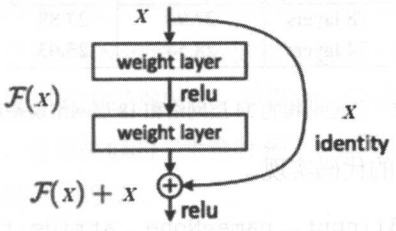


Figure 2. Residual learning: a building block.

图 4-33　残差学习中“快捷连接”（shotcut）结构

这里通过添加“快捷连接（shortcut connections）”将输入信息直接传递到输出层上。快捷连接是指一种跳过一个或多个层次的连接。在我们的案例中，所使用的快捷连接仅仅是进行恒等映射（identify mapping，x identity 就是输入 x 的值），它们的输出被加入到堆叠层次的输出中（图 4-32）。快捷连接的增加并没有引入新的参数，

也没有增加计算复杂度。整个网络仍然可以通过 SGD 反向传播进行端到端的训练。

那么结果如何呢？研究者在 18 层和 34 层网络上做了交叉比对（如图 4-34 所示）。

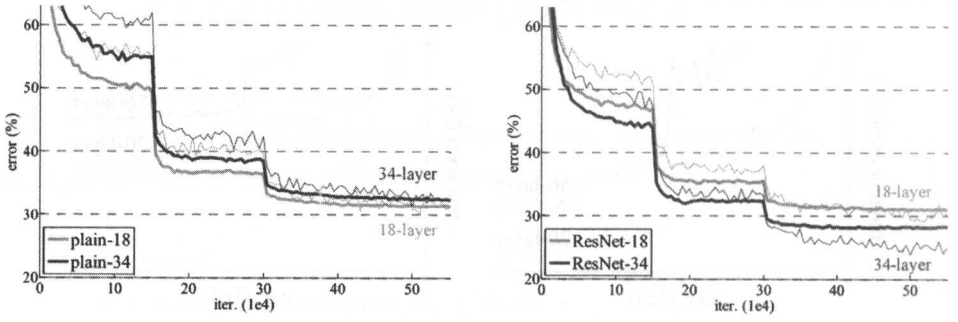


图 4-34　残差结构的 34 层网络和 18 层网络误差对比 1

从图上可以看到两点提高：

（1）34 层的残差网络的效果优于 18 层的残差网络(提高了约 2.8%)。也就是对层数越多的网络残差学习效果越好。

（2）相比于不带残差的网络，34 层的残差网络降低了约 3.5%的最大错误率（如图 4-35 所示），说明残差网络确实对抑制错误率有较好的效果。

	plain	ResNet
18 layers	27.94	27.88
34 layers	28.54	**25.03**

图 4-35　残差结构的 34 层网络和 18 层网络误差对比 2

下面看一下残差网络的代码实现。

```
    def resnet_stage3(input, name=None, stride_flag=True):
        x = input
        if stride_flag:
            conv1 = Conv(input, 256, kernel=(1, 1), stride=(2, 2),
name='%s_conv1_1*1'%name)
        else:
            conv1 = Conv(input, 256, kernel=(1, 1),
name='%s_conv2_1*1'%name)
```

```
        conv2 = Conv(conv1, 256, kernel=(3, 3), pad=(1, 1),
name='%s_conv_3*3'%name)
        conv3 = Conv(conv2, 1024, kernel=(1, 1),
name='%s_conv_1*1'%name)

        res = conv3 + x
        bn = mx.sym.BatchNorm(data=res, name='%s_batchnorm' % name,
fix_gamma=True)
        relu = mx.sym.Activation(data=bn, act_type='relu',
name='%s_relu'%name)
        return relu
```

注意这一句：res = conv3+x。其中 x 就是本网络模块的输入信息，conv3 就是卷积后的取得相应特征向量。再将 x 与 conv3 向量相加，即得到一个输出，供下一层网络调用。

以上就是实现残差网络模块的过程，非常简单，带来的好处却很多，神经网络里有很多类似这样的“小改动大收益”。

4.2.5 十全大补药：通用的提高精度的方法

英特学习到这里发现图像算法大多都以提高精确度为目标，那么如何在现有的模型基础上提高精度呢？现在的 CNN 网络，英特觉得已经优化得不能再优化了，普通人还能再提高吗？一些武侠小说中经常出现一种药物，只要吃下去，加以运气就能提高功力，在模型的世界中，也有类似的“十全补药”，即不改动基础网络结构，而用两种小技巧提升模型整体精度的方法。

1. 联合模型

如果读者参加过 ImageNet、Kaggle 竞赛，对当时第一名的模型名称应该还有些印象，最近几年获得比赛第一名的都叫联合建模或者联合模型（model ensemble），包括当年的 VGG, GoogleNet, ResNet 都用过这个名称。那么这个联合模型是何方神圣？联合模型其实是一个统称，在没有获得好名次时，叫什么都无所谓。但对于研究算法的我们来说，可以将联合模型划分成两种形式：

（1）将深度网络视为单一基类分类器，用基类分类器可以用组合，堆叠，混合，动态选择等多种方法构建混合多重分类器系统，但请记住，只是单纯地将网络添加到你的算法结构中并不会提高精度。

（2）使用深度网络来提取特征，并使用 n-1 层的输出作为分类器的输入。在你使用预训练模型时这个方法特别有用。

我们来重点讨论第二种方法：这种方式有点像 CNN 中的多层次多维度提取特征，只不过 CNN 靠的是卷积层完成多维度提取，这里则将每个基础模型当成一层卷积层看待，做横向的排列组合。仍旧以表情识别为例，我们尝试以三个模型为基础模型，在 n-1 层做特征合并，最后输出表情分类，网络结构如图 4-36 所示。

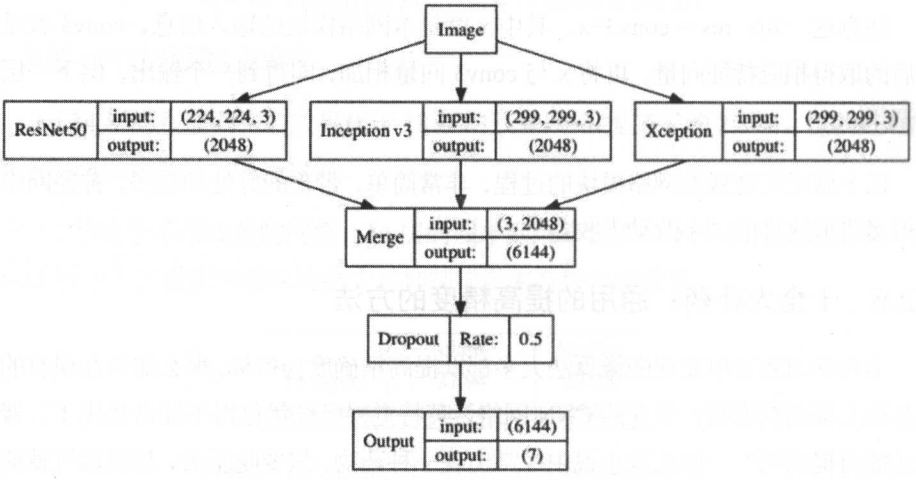


图 4-36　联合模型结构图

我们的表情有 7 类标签，在最后一层用包含 7 位的一维向量 One-Hot 形式来表示并输出。

看到算法结构图可以仔细分析下，为什么要用这三个模型？或者换句话说，为啥要用这三个模型组合一起？

首先，用什么模型实际上是根据图片特点决定，而这里选用这三个模型是因为：

（1）ResNet50 就是在 ResNet152 上的简化版本，共有 50 层，其中网络结构除了

残差结构外，其他部分和 Inception 系列有很多相似的地方。

（2）Xception 作者是 Keras 的作者[1]在 2016 年发表的一篇论文中提到的结构，从名称上就可以看出，也是在 Inception 结构上改进而来。

这样我们发现了这三种结构其实是同源的，它们的基础小架构里有很多相似的地方，但也有一些不同的地方，这就可以让模型在比较接近的层面提取特征，而又能提取到不同的特征。这就是设计的第一个初衷，**基础网络架构要找既有相似的又不能完全一致的，目标是尽可能提取不同微差异特征**。有读者要问，如果用同一种网络结构行不行？比如把上面三种网络结构全部统一成 ResNet50 行不行？这种思想虽然极端，但也是可行的。当然为了差异性，需要在预训练时给出不同的初始化权重，训练的图片也尽可能不要完全重复（允许有交叉），最终得到三个有小差异的 ResNet50 模型。

选择完这三个模型后，英特就用基础图片进行单个网络的预训练了，待预训练完成后，用图中最终模型训练前文提到的 6+1 类表情图片。这里多说一句，预训练完成后，我们的最终模型是由三个特征提取器特征并联层+dropout 层+输出层组成。这个最终模型只有三层，至于 ResNet、Inception v3、Xception 预训练模型则全部固定好参数，只做特征提取用，这时候其中的参数不再发生变化。

最后用最终模型评测人脸表情测试集，val_acc 从 58%提高到 63%，提高的幅度相当大。该算法思路简单、可操作性强，几乎可以用在任何需要提高模型精度的地方，是你手中的强大武器。除了这个武器以外，我们还有再深挖模型潜力的方法吗？请继续往下看。

2. DSD 训练方法[2]

DSD 全称是 Dense Sparse Dense，从字面上理解就是从一个正常的网络结构到稀疏网络结构，最后再回归到正常网络结构的方法，也可以说是“密集-稀疏-密集”的训练方法（如图 4-37 所示）。

---

1 Xception: *Deep Learning with Depthwise Separable*, Fran cois Chollet。

2 本节图片摘自 *DSD: Dense-Sparse-Dense Training for Deep Neural Networks*。

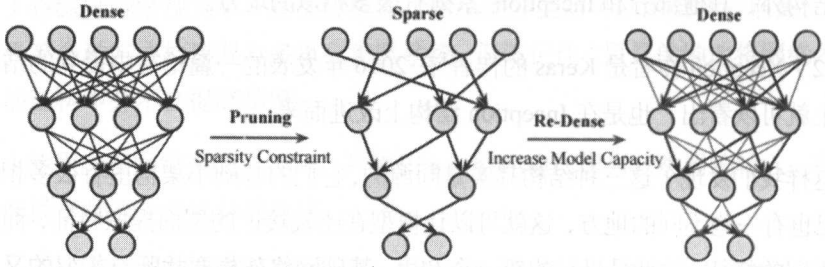


图 4-37　DSD 训练方法

这种训练方法具体步骤如下。

（1）初始的完整网络训练（Dense）：完全按照普通训练规则，比如我们训练一个 InceptionV3 网络，训练 30 次，LR 等参数不做改变。这次训练的目的是找到需要剪枝的单元或权重。

（2）剪枝后网络训练：也就是稀疏结构（Sparse）训练根据第一步的结果，选择需要剪枝的单元或权重去除，剪枝后再次进行网络训练，这步将得到剩余（稀疏）网络结构的权重。

（3）再恢复到完整网络训练（Dense）：将所有剪枝的单元权重恢复，初始赋值为 0，再次进行训练。

以上就是整个 DSD 训练过程，不涉及参数的调整，也就是说在不调整网络参数下，仅仅改变训练的方式，我们就有一个精确度的提高，那么能提高多少呢？请看图 4-38。

Neural Network	Domain	Dataset	Type	Baseline	DSD	Abs. Imp.	Rel. Imp.
GoogLeNet	Vision	ImageNet	CNN	31.1%[1]	**30.0%**	1.1%	3.6%
VGG-16	Vision	ImageNet	CNN	31.5%[1]	**27.2%**	4.3%	13.7%
ResNet-18	Vision	ImageNet	CNN	30.4%[1]	**29.2%**	1.2%	4.1%
ResNet-50	Vision	ImageNet	CNN	24.0%[1]	**22.9%**	1.1%	4.6%
NeuralTalk	Caption	Flickr-8K	LSTM	16.8[2]	**18.5**	1.7	10.1%
DeepSpeech	Speech	WSJ'93	RNN	33.6%[3]	**31.6%**	2.0%	5.8%
DeepSpeech-2	Speech	WSJ'93	RNN	14.5% [3]	**13.4%**	1.1%	7.4%

图 4-38　各网络使用 DSD 训练方式的提高幅度

从图中结果来看，平均有 1%～2%的提高，而且这是一个通用方法，对所有的模型都有效。

我们测试 DSD 方法后，给出面向初学者的简易方法，这里就称它为 S-DSD（Simple-DSD）。所谓简化，就是去除剪枝后的网络训练步骤，也就是剪枝后立即恢复单元权重，进行训练，将 3 次训练变为 2 次训练。由于我们不需要第二步的训练，也就不需要真正进行剪枝，而是仅仅将需要剪枝的网络单元权重赋值为 0。所有过程重新安排如下。

（1）初始的完整网络训练（Dense）：完全按照普通训练规则，训练的目的是将找到需要剪枝的单元或权重。

（2）再恢复到完整网络训练（Sparse-Dense）：将第一步找到的剪枝单元权重赋值为 0，再次进行训练。

我们使用 DSD 方法测试人脸表情识别，演示该方法（S-DSD）同样也可以提高模型精度，以人脸表情 Inception v3 模型为基础模型，在这个模型上应用 S-DSD 方法，来看看以下测试结果：

以下为 Inception v3 初始模型（Dense）测试结果。

```
('Test score:', 1.7123499890523297)
('Test accuracy:', 0.5912388392857143)
('Time:', 7.203636169433594)
```

以下为 Inception v3 初始模型（Sparse，这步并没有训练，只是测试）剪枝测试结果。

```
('Test score (prune):', 2.1974673418640789)
('Test accuracy (prune):', 0.52853528254565674)
('Time (prune):', 6.761786937713623)
```

以下为 Inception v3 初始模型（Dense）剪枝后再恢复初始模型的测试结果。

```
('Test score (prune train):', 1.1673994103207292)
('Test accuracy (prune train):', 0.60612876019955086)
('Time (prune train):', 6.7461981773376465)
```

以上剪枝时的绝对值阈值简单地取为 0.02，读者可以根据自己的模型情况调整阈值或改变取阈值的方式。可以看出模型的 acc 从初始 0.5912 提高到 0.6061，acc 提高

了有 0.015!

为保证我们测试结果的有效性，防止模型因为训练次数增多而提高了 acc，特地在初始模型的基础上验证，继续训练了 16 次，这个训练次数等同于剪枝后恢复初始模型的训练次数，来看下面的结果。

```
('Test score:', 1.1389612658559387)
('Test accuracy:', 0.59319651393297856)
('Time:', 6.806051015853882)
```

结果从 0.5912 到 0.5931，基本上在误差范围内，说明即便使用原始的训练方法继续训练也无法提高模型的精度了。

这一节里，我们谈到两种提高模型精度的方法，第二种方法虽然相对复杂一些，但比起动辄修改模型结构，将深度加 20 多层也要简单很多。希望读者能掌握好这两种秘密武器：十全补药。

4.2.6 图像训练需要注意的地方

1. 训练中的主要参数解释

卷积网络的初学者很容易对参数了无头绪。下面以 MXNet 为例看看卷积网络在训练时的主要参数。

```
def get_iterator(args, kv):
    data_shape = (3, args.data_shape, args.data_shape)
    train = mx.io.ImageRecordIter(
        path_imgrec = train_prefix_path + '.rec',
        data_shape  = data_shape,
        batch_size  = args.batch_size,
        rand_crop   = True,
        rand_mirror = True,
    )

    val = mx.io.ImageRecordIter(
        path_imgrec= val_prefix_path + '.rec',
```

```
        rand_crop   = False,
        rand_mirror = False,
        data_shape  = data_shape,
        batch_size  = args.batch_size,
    )

    return (train, val)
```

下面这两个参数一个是随机修剪（rand_crop），一个是随机镜像（rand_mirror），也就是一个做图片的分割，一个是随机地对图片做些翻转，主要是为了增加图片的多样性，增加训练样本，避免一些训练集比较小的问题。我们在下一小节中将详细叙述如何增加训练样本。

```
rand_crop   = True,
rand_mirror = True,
```

接下来看 args 参数，第一个需要注意的是确定网络结构参数。

```
parser.add_argument('--network', type=str, default='resnet152',
                    choices = ['alexnet', 'vgg', 'googlenet',
'inception-bn',
                               'inception-bn-full', 'inception-v3',
'inception-v4', 'resnet', 'inception-resnet-v2', 'resnet152'],
                    help = 'the cnn to use')
```

因为本次英特要训练的图片分辨率都比较大，基本在 299×299 的两倍以上，综合考虑准确率等因素，我们选择 Inceptions v3 网络结构。

```
parser.add_argument('--lr', type=float, default=.01,
                    help='the initial learning rate')
```

第二个需要注意的是 lr 参数，lr 指 learning rate，也就是学习速率，它是模型调参的关键因素。

```
parser.add_argument('--clip-gradient', type=float, default=5.,
                    help='clip min/max gradient to prevent extreme
value')
```

一次迭代中权重的更新过于迅猛的话，很容易导致 loss divergence，clip_gradient 的直观作用就是把权重的更新限制在一个合适的范围。

```
parser.add_argument('--num-epochs', type=int, default=10,
                    help='the number of training epochs')
```

上述代码中 epochs 是训练的迭代次数，指明需要训练多少次。

```
parser.add_argument('--num-examples', type=int, default=24371,
                    help='the number of training examples')
```

num-examples 参数用于确定图片总数量。

```
parser.add_argument('--num-classes', type=int, default=5,
                    help='the number of classes')
```

神经网络的输出都需设定分类标签，上述代码指明有分类标签的数量。

```
parser.add_argument('--data-shape', type=int, default=299,
                    help='set image\'s shape')
```

上述代码用 data-shape 设置输入图片分辨率。

最后，我们使用以下代码进行训练。

```
# train
train_model.fit(args, net, get_iterator)
```

当训练完成后，我们就能用 evaluation 函数或 predict 函数进行模型的评估或预测新的数据了。

2. 数据的预处理和数据提升

图像训练中，我们不可避免地会遇到训练数据过少的情况，而训练数据过少又在很大程度上制约我们训练的准确性，如何找到一种成本最低的方法增加训练图片样本呢？

对于这个问题各种深度学习框架都有自己的一些处理方式，总体思路一致：靠着变换图像的大小，颜色，主体的位置等等来将一幅图像增加到多幅图像，这样相当于

训练数据扩展了 n 倍。

比如在上一节说到 MXNet 里有 rand_crop = True，随机修剪图片，在图片小于或大于输入大小时有效。rand_mirror = True，随机地水平翻转图像。

而 Keras 里也有类似设置，代码如下。

```
datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')
```

其中，

- rotation_range 是一个 0~180 的度数，用来指定随机选择图片的角度。
- width_shift 和 height_shift 用来指定水平和竖直方向随机移动的程度，这是两个 0~1 之间的比例。
- rescale 需要特别注意，从名字上看是做缩放，实际上是对图像的颜色值做归一化。一般来说，输入的图像颜色值范围是 0~255 的整数，直接将这些值输入将会导致图像的值过高或过低，所以我们这里使用 rescale 将它归一化到[0,1]区间，因此这个 rescale = 1/255
- shear_range 用来处理剪切变换的程度。
- zoom_range 用来进行随机的放大。
- horizontal_flip 随机地对图片进行水平翻转，这个参数适用于水平翻转不影响图片语义的情形。
- fill_mode 用来指定当需要进行像素填充，如旋转、水平和竖直位移时，如何填充新出现的像素。

其他深度学习框架中是不是也有类似的设置？带着这个疑问我们以 TensorFlow

为例来看看其封装的函数 inception_preprocessing.py，抽取其中的几句来看：

```
    image = tf.image.random_saturation(image, lower=0.5,
upper=1.5)
    image = tf.image.random_brightness(image, max_delta=32. / 255.)
    image = tf.image.random_contrast(image, lower=0.5, upper=1.5)
    image = tf.image.random_hue(image, max_delta=0.2)
```

上述代码依次表示对饱和度、亮度、对比度和色调进行随机变换并扩展图片。基本上主流的深度学习框架都有相关的图像设置部分，尽量不让你的时间花在一般性的图片处理上。

3. 防止过拟合

过拟合是什么？简单来说就是在训练时 train_acc 很高，在做新图片测试时 val_acc 非常低，这个差距有时候会是 acc1>acc2×2，导致过拟合的原因有不少，大家先记清楚这种现象，如果在训练集上拟合较好，acc 较高，一旦到测试集上 acc 较低就是过拟合了[1]。

Dropout 的发明就是为了缓解过拟合问题（注意：过拟合是常见现象，跟你的训练集、网络结构都高度相关，Dropout 作为手段只能起到缓解作用），那么什么是 Dropout 呢？

Dropout 是指在模型训练时随机让网络某些隐含层节点的权重不工作，不工作的那些节点可以暂时认为不是网络结构的一部分，但是它的权重得保留下来（只是暂时不更新而已），因为下次样本输入时它可能又得工作了。

在训练过程中，随机地使一些神经元输出为 0，使其失效，这样有可能可以获得更多的特征表达，每一个神经元不完全依赖于其他的神经元。

图 4-39 左侧为一个标准的全链接的神经网络，右侧为我们使用 Dropout 机制后，人为地随机造成几个神经元失效。

1 定义不太严谨，这里为方便理解，不探讨其他情况。

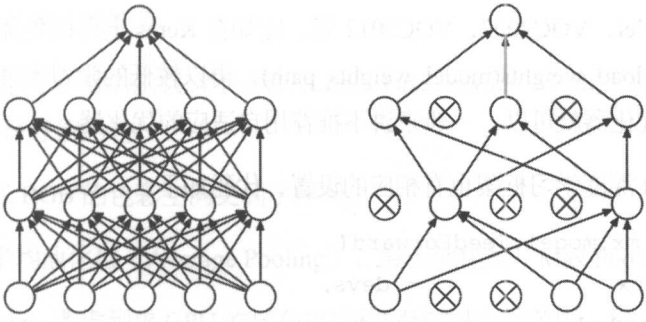


图 4-39 Dropout 示意图

加入 Dropout 并调整参数使之适应网络，那么如何判断 Dropout 有效果呢？这就要看 val_acc 了。在应用 Dropout 后，虽然 train_acc 下降，但 val_acc 上升了 3%，排除其他影响因素，我们认为 Dropout 的加入一定程度地提高了泛化能力并缓解了过拟合。

4．使用预训练网络来做微调（fine-tune）

对一个做图片分类的神经网络来说，要达到可用的程度，一般训练的每一类别图片量级至少在 5000～10000 张左右。如果我们没有这么多图片，甚至加入前文探讨的数据提升的方法也凑不齐这些图片，但我们又非常需要训练这个网络怎么办？这时候就需要搬出我们的预训练网络了，一般的预训练网络都是指在几十万张图片训练集上取得一个较好效果的图片多分类器，然后用我们现有的图片在这个预训练网络的基础上继续训练，利用现有的图片继续训练就是为了让网络重新适应现有的图片训练集，这种适应过程往往比从头训练一个网络更快而且更好。利用预训练模型的方法和迁移学习的思想很像。

使用预训练网络来做微调，是利用在大规模数据集上预训练好的网络，将网络的参数应用到我们的模型中。因为这样的网络已经被验证过，在多数图片上能够取得不错的特征，利用这样的特征可以让我们获得更高的准确率。也就是我们站在“巨人”的肩膀上让模型更进一步，更可以在一定程度上规避训练集过少的问题。

所有的深度学习框架都提供了预训练方法，并提供了一些大数据集上训练好的模

型比如 ImageNet，VOC2007，VOC2012 等。比如在 Keras 上可以先加载另一个网络的权重 model.load_weights(model_weights_path)，再以较低的学习率进行训练，这时候选用 SGD 优化器就可以，一般来讲不推荐用自适应的优化器。

而 MXNet 深度学习框架也有相应的设置，代码如下。

```
model = mx.model.FeedForward(
        ctx                = devs,
        symbol             = network,
        num_epoch          = args.num_epochs,
        initializer        = mx.init.Load(params,default_init =
mx.init.Xavier(factor_type="in", magnitude=2.34)),
        optimizer          = optimizer,
        **model_args)
```

请注意这句，它的本意是可以在初始化时加载参数，当然也可以用于加载一个预训练网络。

```
    initializer  = mx.init.Load(params,default_init =
mx.init.Xavier(factor_type="in", magnitude=2.34)),
```

如果要在 initializer 加载预训练网络，可以按下面的办法来操作。

```
    initializer        =
mx.init.Load('../tmp/model/Inception-7-0001.params',default_init
mx.init.Xavier(factor_type="in", magnitude=2.34)),
```

现实的大部分应用基本上都要用到一个预训练网络（基础模型）做 fine-tune，因为大型数据集训练太过耗时，而且没必要每次都单独训练，并且在训练好的模型上做 fine-tune 能将准确度提高到新的程度，所以推荐读者这样操作。

5. 他山之石[1]

除了以上提到的经验外，前人对于图像训练的优化已经总结了不少注意事项，下

1 本小节部分引用论文 Dmytro Mishkin，*Systematic evaluation of CNN advances on the ImageNet*，2016。

面看看这些注意事项的细节。

① 能使用 ELU 激活函数就不要使用 BN（batch norm）；而如果能使用 ReLU 激活函数，那就最好使用它。

② 转换成 RGB 的色彩空间更利于学习。

③ 使用平均池化层（Average Pooling）、最大池化层（Max Pooling）的和值。

④ batch size：考虑到单 GPU 的显存容量和网络结构，该范围一般定于 128～256。如果 batch size 超过 256，就需成比例地降低学习率（lr），以取得最佳准确率。batch size 和 lr 的关系可以使用公式表达： lr=0.01 × batch_size/256

⑤ 一般使用全连接层用于最后输出。

⑥ 训练数据的干净程度比数据大小更重要。

⑦ 如果不能增加图像尺寸，减少卷积层的 stride 也可以达到同样的提取特征的效果。

⑧ 如果你的网络类似于 GoogLeNet 这样的高复杂性网络，尽量不要多做修改。

⑨ 学习率公式，论文中对比测试使用线性函数的准确率最高，lr $=L_0$(1-i/M)。

好，知道这些规则之后，我们需要知道为什么会有这些规则，我们一条一条地来梳理。

① ELU 激活单元属于完全非线性激活公式 $y=x$, if $x>=0$, else $\alpha(e^x-1)$，而 ReLU 属于分段线性激活，$y=\max(x,0)$,在我们使用完全非线性公式时，对于每一层的归一化并不敏感，但分段线性函数比较敏感，最好用归一化过程也就是 BN(batch norm) 做标准统一。SoftPlus 也属于非线性函数，在 ELU 没有出现时，softplus 常常用于非线性激活，类似于人的神经细胞工作机制。

② 本条规则是实验得出的数据，在论文中 RGB 色彩空间得到最大的提高。个人猜测是 RGB 色彩空间能更准确地保持图片的信息,如图 4-40 所示。可以看到图中以 RGB 为基准，其他色彩空间都有负增长。

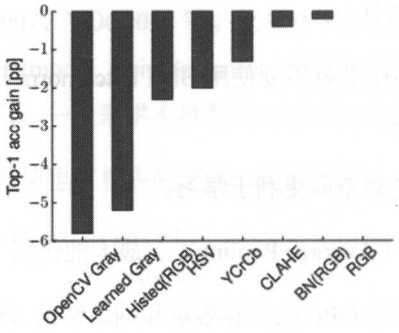


图 4-40　色彩空间对于最终准确度的影响

③ 使用 Average Pooling 加 Max Pooling 的和是为了尽可能广地提取特征，并与提取的最大特征合并，从而更好地表达原图像。

④ 也就是 batch size 越大，lr 值取得越小，这样能取得和小 batch size 接近的准确率。

⑤ 最后一层一般用全连接层，也就是 FC 来作为输出层。

⑥ 训练图片如果有噪音的话（这里指训练图片的标记不准确），尽量去除这些噪音。噪音的影响非常大，如果我们使用 20w 无噪音的图片和 50w 有噪音图片（包含大于 20w 无噪音图片）分别训练，其准确度值 top-1 acc 基本一致，如图 4-41 所示。

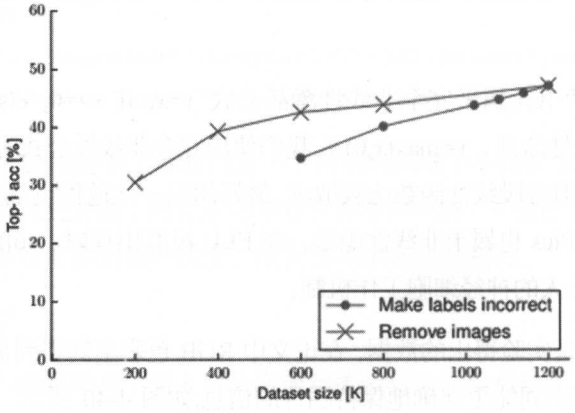


图 4-41　噪音训练集和无噪音训练集的准确度对比

⑦ 这条很好理解，卷积核移动间隔减小，意味着提取的特征更多。

⑧ 尽量只改复杂网络的头几层和最后输出层，可以对激活单元进行改动，如非特殊任务，尽量不再做其他修改。

⑨ 公式：lr $=L_0(1\text{-}i/M)$ ，其中L_0是当前的学习率，M 是迭代总次数， i 是目前的迭代次数。

以上只是论文作者根据某些测试给出的对比结果，这些规则随着时间的推移和网络的优化一直在变化中，读者可以参考，但无需盲从。

4.3 目标检测[1]

英特在完成图片的分类项目后，公司又想在实时追踪领域进行一些尝试，而图片分类器都是对完整的一张图片做分类，要求输入的是完整的图片，并不对其中的物体进行框定，而实时追踪必然需要对物体进行框定和识别。这就进入了深度学习之图像领域的第二个方向：目标检测。

英特在看了一些文章之后，觉得目标检测经历了可以清楚划分的三个阶段。

第一个阶段：传统思路

什么是传统思路？如图 4-42 所示。

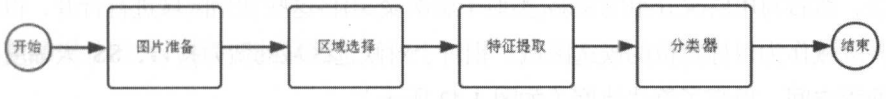


图 4-42 目标检测传统思路

（1）图片准备：对图片进行灰度化，颜色变化，饱和度变化等。

（2）区域选择：这一步是为了对目标的位置进行定位。由于目标可能出现在图像的任何位置，而且目标的大小、长宽比例也不确定，所以最初采用滑动窗口的策略对整幅图像进行遍历，而且需要设置不同的尺度，不同的长宽比。这种穷举的策略虽然包含了目标所有可能出现的位置，但是缺点也是显而易见的：时间复杂度太高，产生

1 本节关于目标检测三个阶段的部分内容改编自 http://chuansong.me/n/353443351445，《基于深度学习的目标检测研究进展》，王斌。

的冗余窗口太多，这也严重影响后续特征提取和分类的速度和性能（实际上由于受到时间复杂度的问题，滑动窗口的长宽比一般都是固定设置的，所以对于长宽比浮动较大的多类别目标检测，即便是滑动窗口遍历也不能得到很好的区域）。

（3）特征提取：传统的模式里用了很多人工方法去提取特征，而在此处最早的人工特征提取也是模拟了人的眼睛识物，比如目标形状的多样性，目标是否契合圆形、方形、多边形，光照的运用，光照明暗变化，背景和前景的变化。这些因素的重叠，又基本基于人工的特征选择，这使得如果要处理单一性问题会比较容易，比如识别车牌，车标等。如果被提取的物体本身就多样性很高，则利用这些特征设计一个鲁棒性高的提取模型就不那么容易。也就是说的它的通用性不是那么好。而提取特征的好坏也直接影响到下一步分类器的输出。

（4）分类器：传统的分类器，这里主要使用了 SVM，Adaboost 等。

传统模式有以下两个问题：一是区域选择没有针对性，导致时间复杂度较高，冗余窗口多；二是在特征选择上对多变化的物体无法提取一批好的通用性特征。

研究者们首先解决的不是提取特征的问题，而是使用 Region Proposal 解决区域选择问题。Region Proposal 的典型方法就是 Selective Search(下面简称 SS)。

SS 方法的思想是先使用图像分割的方法得到一些初始分割区域，有点像超像素概念，然后利用层次分组的策略(类似于层次聚类)将这些初始区域进行合并，得到的这些区域作为目标定位的候选区域。相对于对候选区域的蛮力搜索，SS 大幅度降低了搜索空间，提高了算法速度（如图 4-43 所示）。

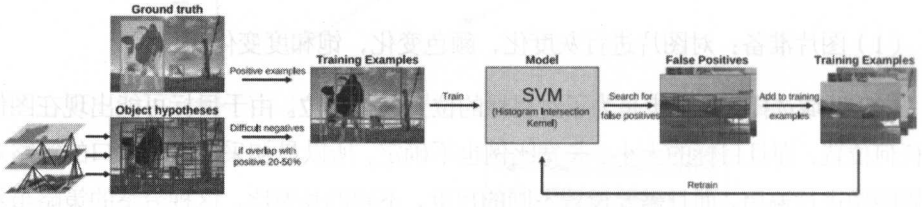


Fig. 3 The training procedure of our object recognition pipeline. As positive learning examples we use the ground truth. As negatives we use examples that have a 20–50 % overlap with the positive examples. We iteratively add hard negatives using a retraining phase

图 4-43　Selective Search 原理图

有了候选区域，剩下的工作实际上就是对候选区域进行特征提取和分类的工作。

第二个阶段：基于 R-CNN 的深度学习目标检测算法

（1）R-CNN。

前文简单提过：R-CNN 就是用 Selective Search 做区域选择，用 CNN 提取图像特征，用 SVM 做分类器（如图 4-44 所示）。

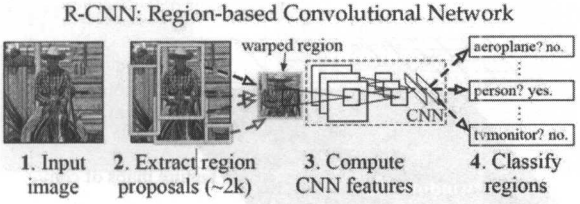


图 4-44 R-CNN 算法流程图

将 R-CNN 主要流程具体化：

① 利用 Selective Search 算法在输入图像中提取 2000 个左右的区域作为 CNN 输入。

② 将每个 Region Proposal 缩放为 227×227 的大小并输入到 CNN。

③ 将每个 Region Proposal 提取到的 CNN 特征输入到 SVM 进行分类。

小结：R-CNN 在 PASCAL VOC2007 上的检测结果从 DPM HSC 的 34.3%直接提升到了 66%（mAP）。

虽然提升很大，但问题也很多。

① 训练分为多个阶段，步骤繁琐：微调网络+训练 SVM+训练边框回归器。

② 训练耗时，因为 2000 多个候选区域都要进行卷积层特征提取，存在大量的重复计算。

针对以上提到的 R-CNN 训练耗时问题，有了下面的解决方案：SPP-NET。

（2）SPP-NET。[1]

1 SPP-NET (ECCV2014, TPAMI2015) (Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition)。

SPP-NET 只对图像提取一次卷积层特征，然后将 Region Proposal 提取的候选区域在原图的位置映射到卷积层特征上，从而节省大量时间（如图 4-45 所示）。

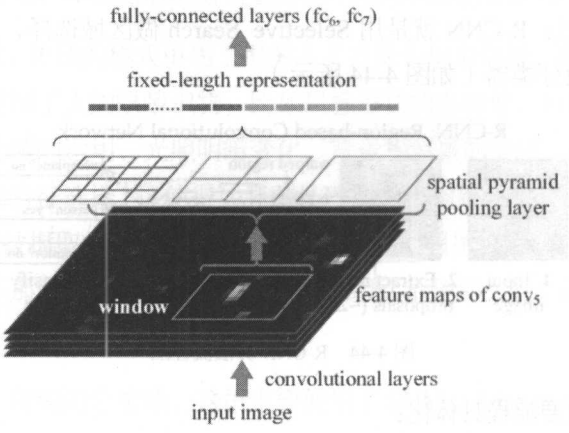


图 4-45　SPP-NET 空间金字塔采样（spatial pyramid pooling）

将每个 window 划分为 4×4，2×2，1×1 的块，然后每个块使用 max-pooling 下采样，这样对于每个 window 经过 SPP 层之后都得到了一个长度为(4×4+2×2+1)×512 维度的特征向量，将这个作为全连接层的输入进行后续操作。

由于 Region Proposal 大小不一，所以需要通过 spatial pyramid pooling layer 层将 Region Proposal 通过卷积层得到的特征映射为固定长度的特征。图中的 window 就是原图一个 Region Proposal 对应到特征图的区域，只需要将这些不同大小 window 的特征映射到同样的维度，将其作为全链接的输入，就能保证只对图像提取一次卷积层特征。

小结：相比于 R-CNN，使用 SPP-NET 可以大大加快目标检测的速度，但是依然存在以下问题。

① 训练分为多个阶段，步骤繁琐：微调网络+训练 SVM+训练边框回归器。

② SPP-NET 在微调网络的时候固定了卷积层，只对全连接层进行微调。

SPP-NET 也有很多问题，不过 R-CNN 作者 RBG[1]单枪匹马地继续更新了 R-CNN

1　Ross B. Girshick。

系列，推出了 R-CNN 改进型，Fast R-CNN（如图 4-46 所示），下面来看看它和 R-CNN 有什么不同。

（3）Fast R-CNN（ICCV2015）。

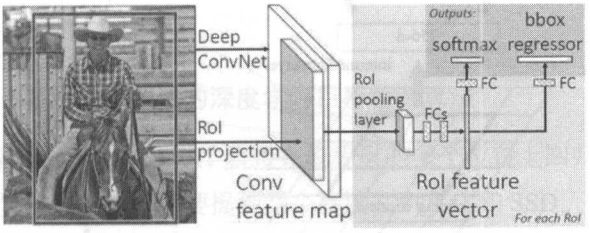


图 4-46 Fast R- CNN 框架图

与 R-CNN 框架图对比，可以发现主要有两处不同：一是最后一个卷积层后加了一个 ROI pooling layer，二是损失函数使用了多任务损失函数(multi-task loss)，将边框回归直接加入到 CNN 网络中训练。

① ROI pooling layer 对应于 SPP-NET 的 spatial pyramid pooling layer，不同之处在于 SPP-NET 使用的是不同大小的金字塔映射，而 ROI pooling layer 通过池化得到一个 $7\times7\times512$ 固定大小的特征向量作为后面全连接层的输入。

② R-CNN 训练过程分为了三个阶段，Fast R-CNN 通过在全连接层后接入 softmax 层和 bbox regression 层，实现了端到端的训练，避免在 R-CNN 中分别训练卷积层和 SVM 分类器。不足之处是 Region Proposal 仍然是一个独立的过程。

Fast R-CNN 相对 R-CNN 而言有所提高，而区域选择这部分，RBG 还是觉得有改进空间，遂和其他研究者一起，给出了改进型 Faster R-CNN。

（4）Faster R-CNN（NIPS2015）。[1]

Faster R-CNN 等于 Fast R-CNN+RPN，是针对 Fast R-CNN 所留下的 Region Proposal 的问题提出的进一步的改进。Faster R-CNN 的核心是 RPN（Region Proposal Networks）和权值共享。

1 Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks。

RPN 的核心思想是使用 anchor 机制直接产生 Region Proposal，其本质上就是滑动窗口，如图 4-47 所示。

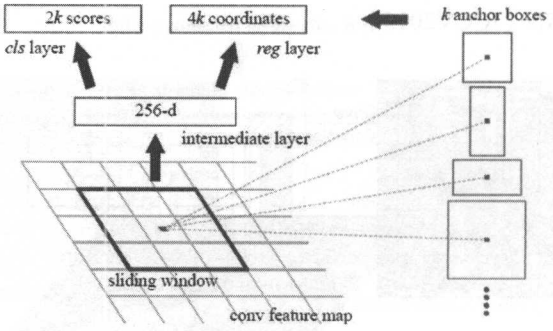


图 4-47　RPN 网络结构图

那么什么是 anchor 机制？图 4-47 在最后一层 conv feature map 上用一个 3×3 的滑动窗口进行卷积，每一个窗口提取 k 个 anchor boxes（这里 k=9，简单讲就是 3 种尺寸和 3 种长宽比的组合），对于每一个 anchor 通过 intermediate layer 提取 256 维的特征，后面接 cls layer 和 reg layer 分别用于分类和边框回归。所以每一次滑动窗口都会输出 $2k$ 个 scores 和 $4k$ 个 coordinates。一张大小 1000×600 的图片大约生成 20000（约为 60×40×9）个 anchor。

所谓权值共享（如图 4-48 所示），就是 RPN 和 Fast R-CNN 共享卷积层的特征，RPN 将 anchor 产生的 score 较高的 boxes 送入 Fast R-CNN 的 ROI pooling 进行分类和边框回归。

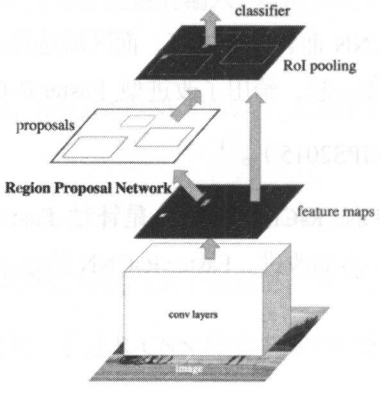


图 4-48　RPN 结构

至此，Faster R-CNN 终于将 R-CNN 三段式的训练过程（Selective Search 提取边框 + CNN 提取特征 +SVM 分类）统一成真正可以端到端的训练，使目标检测的流程越来越精简，速度也越来越快，同时精度也越来越高。但是，即便是 Faster R-CNN，速度只有 5fps[1]，无法胜任实时目标检测的需求，于是出现了基于另一种思想的目标检测的算法：基于回归。

第三个阶段：基于回归方法的深度学习目标检测算法

所谓回归，就是给定输入图像，直接在卷积层的多个位置上回归出这个位置的类别，目标边框和置信度。这里主要提到两个算法：YOLO 和 SSD。

（1）YOLO（CVPR2016, oral）[2]。

YOLO 和 Faster R-CNN 一样，是一种端到端的训练过程，不同之处在于 YOLO 直接在特征层上回归出类别和边框，从而节省了大量提取边框的时间，速度可以达到实时检测的条件。

具体来说，YOLO 将输入图像划分为 S × S 个网格，如果一个物体的中心落在某网格内，则相应网格负责检测该物体。在训练和测试时，每个网格预测 B 个 bounding boxes，每个 bounding box 对应 5 个预测参数，即 bounding box 的 4 个坐标加一个置信度。最后用非极大值抑制算法（NMS）去除冗余的窗口即可，过程和思想都非常简单（如图 4-49 所示）。

这里 s=7，voc 上有 20 个类别，所以每个网格输出（4+1）×2+20=30 维的向量，总共要输出 7×7×30 维的张量（如图 4-50 所示）。

小结：YOLO 将目标检测任务转换成一个回归问题，大大加快了检测的速度，使得 YOLO 可以每秒处理 45 张图像。但是 YOLO 也存在问题：没有了事前区域选择机制，只使用 7×7 的网格回归会使得目标不能非常精准的定位，这也导致了 YOLO 的检测精度并不是很高[3]。针对 YOLO 检测精度问题，研究者们又拿出下一个兼具效率

1 论文摘要中给出的这个数据。

2 YOLO：You Only Look Once， Unified, Real-Time Object Detection。

3 2016 年 12 月的论文提出 YOLO v2 结构，特别改进了 YOLO v1 的问题，测试中精度达到 SSD 500×500 的精度，而且速度相对于 YOLO v1 还有提高。

和精度的算法：SSD。

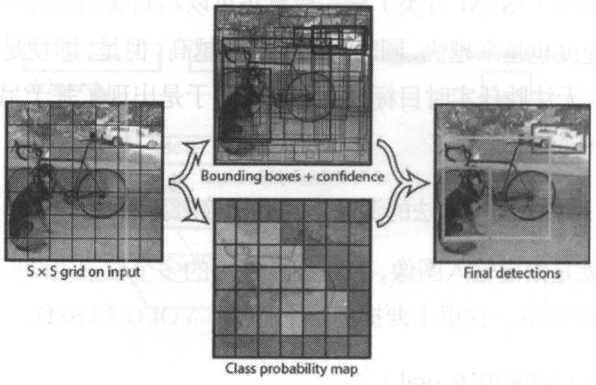


图 4-49　YOLO 的目标检测的流程图

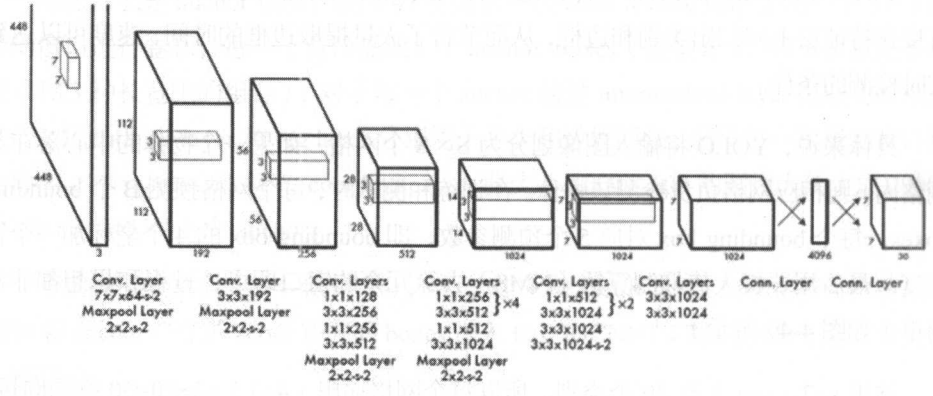


图 4-50　YOLO 算法结构图

（2）SSD[1]。

SSD 结合了 YOLO 的回归思想以及 Faster R-CNN 的 anchor 机制，或者说 SSD=YOLO + RPN，从而有效解决了 Faster R-CNN 速度慢和 YOLO 精度不准的问题。

SSD 获取目标位置和类别的方法跟 YOLO 一样，都是使用回归，但是 YOLO 预测某个位置使用的是全图的特征，而 SSD 对 anchor 提取的边框进行局部回归（如图 4-51（c）所示）。

1　SSD: Single Shot MultiBox Detector。

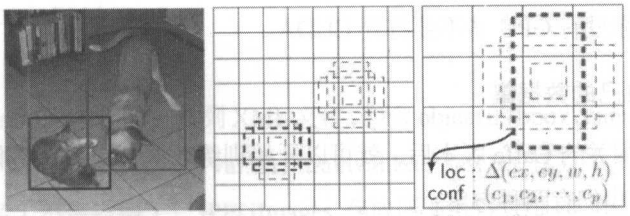


图 4-51 SSD 框架图

总结：SSD 结合 Faster R-CNN 和 YOLO 的优点，在保证检测精度的条件下，GPU 上的速度已经可以达到 58 秒每帧，使得对实时视频流的检测处理变为可能。无人驾驶及很多工业机器人对算法精度和效率都有要求，所以精度和效率的提升一直是目标检测提升的两个重要方面。

目标检测算法已介绍完，而英特已准备好了 SSD 开源代码，训练自己的数据，并使用训练好的模型做一些测试，看看效果，我们一起看看整个过程。

4.3.1 用 SSD 来实现目标检测应用

1. 安装 SSD

SSD 项目主页：https://github.com/weiliu89/caffe/tree/ssd。

首先，我们把项目代码 clone 下来，然后编译。

```
git clone https://github.com/weiliu89/caffe.git
cd caffe
git checkout ssd
```

GPU 版安装需修改 Makefile.config 文件， 修改完成后：

```
Make;make py
```

到这里就完成了 SSD 的安装，接下来是如何训练自己的数据集。

这里有以下需要注意的地方：

（1）在编译时会要求安装 OpenCV，请注意版本号，安装 2.4.10 或 3.0 后的版本。

（2）把 Makefile.config 文件中有关 python_layer 的那一行改为 1。

（3）如果你没有 GPU 请不要打开 CUDA。

2. 训练自己的数据集

在安装完相关的 Caffe 版本后，就可以开始训练数据。首先需要准备好数据集目录，在 SSD 中我们有两种方案：

第一，保持原来的文件目录结构及文件名不变，只替换里面的数据。

第二，重新建一个与之前类似的目录结构，改成自己命名的文件夹。

第二种方法，因为待修改程序里涉及数据路径的代码，所以务必仔细检查。我们采用第二种方案。

在/data 目录下创建一个自己的文件夹：（现在项目新建了一个 lavector 文件夹）

```
cd /data
mkdir mydataset
```

把/data/VOC0712 目录下的生成数据的脚本文件 create_list.sh 、create_data.sh、labelmap_voc.prototxt 这三个文件拷贝到/mydataset 下：

```
cp data/create* ./mydataset
cp data/label* ./mydataset
```

其中 labelmap_voc.prototxt 定义了标签类别。

在/data/VOCdevkit 目录下创建 mydataset，并放入自己的数据集。

```
"""
cd data/VOCdevkit
mkdir mydataset
cd mydataset
mkdir Annotations
mkdir ImageSets
mkdir JPEGImages
cd ImageSets
mkdir Layout
mkdir Main
```

```
mkdir Segmentation
"""
```

其中Annotations中存放一些列XML文件，包含object的bbox，name等；ImageSets中三个子目录下均存放 train.txt，val.txt，trainval.txt，test.txt 这几个文件，文件内容为图片的文件名（不带后缀）；JPEGImages 存放所有的原始训练图片；在/examples下创建 mydataset 文件夹 mkdir mydataset，在文件夹内存放生成的 Imdb 文件。

上述文件夹创建好后，开始生成 imdb 文件，在创建之前需要修改相关路径，执行下列命令：

```
./data/mydataset/create_list.sh
./data/mydataset/create_data.sh
```

此时，在 examples/mydataset/文件夹下可以看到两个子文件夹，mydataset_trainval_lmdb, mydataset_test_lmdb；里面均包含 data.dmb 和 lock.dmb;

至此，数据集就做好了，接下来就开始训练了。训练程序为/examples/ssd/ssd_pascal.py，运行之前，我们需要修改相关路径代码：

cd /examples/ssd

vim ssd_pascal.py

修改如下：

第 57 行： train_data 路径；

第 59 行：test_data 路径；

第 197—203 行：save_dir、snapshot_dir、job_dir、output_result_dir 路径；

第 216—220 行： name_size_file、label_map_file 路径；

第 223 行：num_classes 修改为类别数+1，这个+1 是保留类别_background；

第 315 行：num_test_image：测试集图片数目；

另外，如果你只有一个 GPU，需要修改 285 行：将 gpus= “0,1,2,3” 改为 “0” ；

否则，训练的时候会出错。

修改完后，开始训练：

python examples/ssd/ssd_pascal.py

3. 测试效果

执行 python ssd_detector.py (根目录下)，可以得到图 4-52 的识别结果。

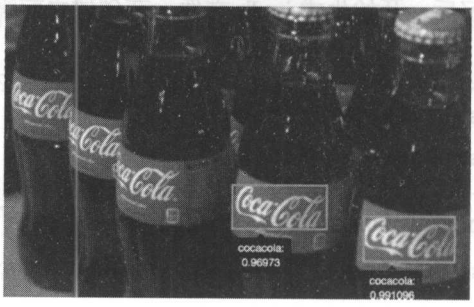


图 4-52　SSD 识别结果

4.3.2　SSD 训练源码提示

前面我们谈了如何安装、训练、测试 SSD，但英特还觉得不止于此，还想更深入地去了解 SSD 的机制，另外在训练时也避免不了要修改原代码以期望获得一个较好的结果。下面我们跟随英特一起了解下与 SSD 训练相关的重要代码。

先找到 ssd_pascal.py 文件，逐行查看，在第 86—87 行，定义了输入网络的图像大小。

```
resize_width = 300
resize_height = 300
```

第 89—174 行，定义了 batch_sampler，它的目的是用图像变换的方式增加样本数量，不只是对原图进行增加，也对我们标记的 object 进行放大缩小的操作，以增加 object 样本。

第 175 行，开始 train_transform_param：训练集图片的扩充。

第 212 行，开始 test_transform_param：测试集图片的扩充。

第 225 行，batch 关闭正则化 use_batchnorm = False。

第 228—232 行，赋值不同的 lr。

```
if use_batchnorm:
    base_lr = 0.0004
else:
    # A learning rate for batch_size = 1, num_gpus = 1.
    base_lr = 0.00004
```

第 272 行，ignore_cross_boundary_bbox = False 是是否忽略交叉框的情况，一般选择不忽略。

第 331 行，定义 gpu，如果有多 gpu 则一并定义。

第 337—347 行，处理多 gpu 时的分配。

第 349—356 行，针对不同的情况得到不同的学习率。

```
if normalization_mode == P.Loss.NONE:
  base_lr /= batch_size_per_device
elif normalization_mode == P.Loss.VALID:
  base_lr *= 25. / loc_weight
elif normalization_mode == P.Loss.FULL:
  # Roughly there are 2000 prior bboxes per image.
  # TODO(weiliu89): Estimate the exact # of priors.
  base_lr *= 2000.
```

第 365—389 行是常规训练参数设置。

第 438 行，AddExtraLayers(net, use_batchnorm, lr_mult=lr_mult)

第 14 行，实现了下面这个函数：

def AddExtraLayers(net, use_batchnorm=True, lr_mult=1),

主要是在用到的基础算法基础上增加多层，做一些改造，这里的基础算法指的是 VGG16 和 InceptionV3 等。

从第 476 行到最后就是参数的设置和模型文件的操作。

至此，目标检测部分就已经全部讲完了，读者可以参考上面步骤自己动手尝试下。下节我们聊聊视觉领域的其他应用。

4.4 视觉领域的应用

4.4.1 艺术风格画

在艺术领域，尤其是绘画领域，艺术家们通过创造不同的内容与风格，并相互交融影响来创立独立的视觉体验。如果给定两张图像，以现在的技术手段，完全有能力让计算机识别出图像的具体内容。那计算机能否像艺术家一样交融几种绘画风格来产生一种新的风格？换个角度来说，风格是一种很抽象的东西，在计算机的眼中，能认识所谓的风格吗？还是只有一堆像素点？

风格的识别，其实是大脑的一种归纳能力，在看过一些名画后，就能很有效地辨别出不同风格的画。怎样让深度学习也具有这样的功能，接近人类对绘画风格的归纳能力，让计算机不但能识别画作，还能将这些风格转移到一张普通照片上，从而使计算机有“复制”名画的本领呢？

这一切需要靠卷积网络来完成。卷积网络具有找出更复杂、更高层特征的能力，我们可以用特征来分类、预测，也可以用来做目标识别，而现在我们要做的是图像风格的提取和融合。

这里的主要思想可以总结为：一个足够深的卷积网络可以在高层表示出图像的高级抽象特征，如果把这些高级抽象特征应用到另外一张图上，那么另外一张图也可以继承这些高级特征（如图 4-53 所示）。

整个网络用一个卷积网络就能实现，输入的是两张图片，一张是风格图片，一张是普通照片，目标是要让普通照片渐渐趋向于风格图片。在网络的特征抽取后，将这种特征转移到输入的普通照片上，就完成了图 4-54 中艺术图画的绘制。

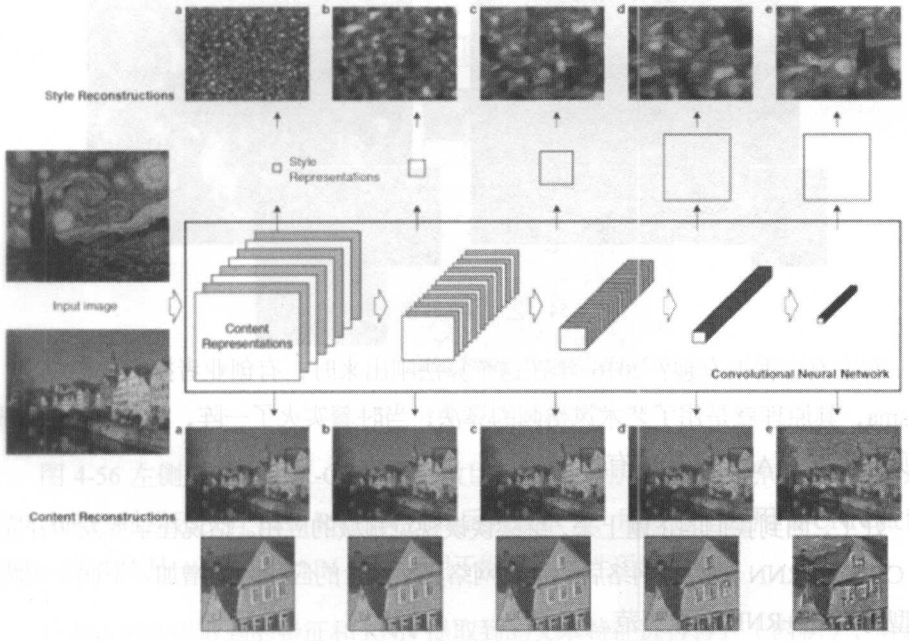


图 4-53 艺术风格算法示意图

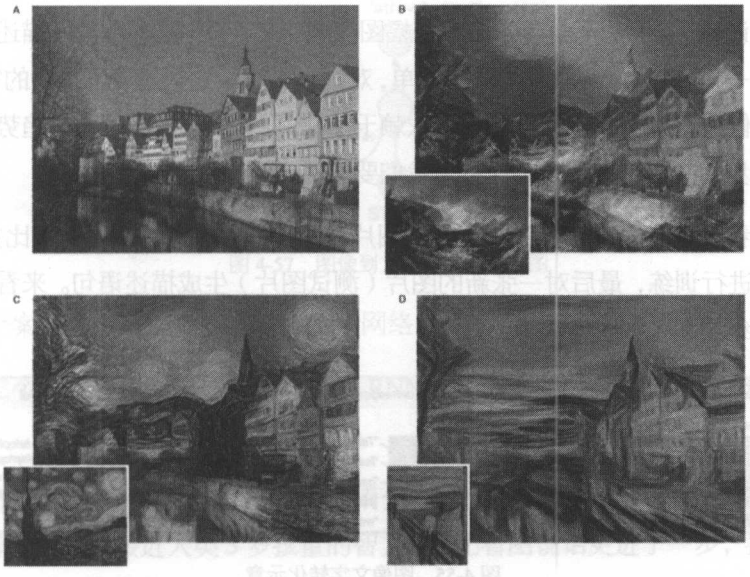


图 4-54 艺术风格转移示例

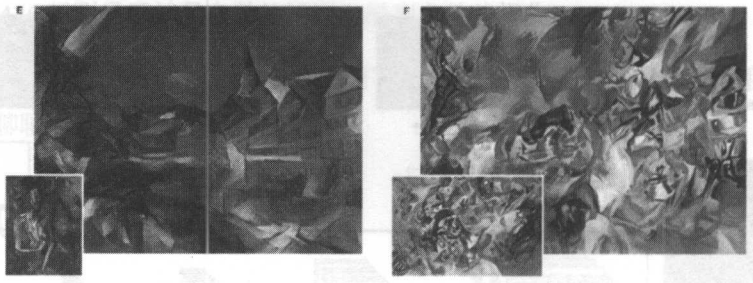


图 4-54　艺术风格转移示例（续）

有没有觉得很有趣？2016 年在这个算法刚出来时，有创业者推出一个 App 叫 Prisma，其原理就是用了艺术风格画的算法，当时着实火了一阵，还顺利地融了资。可见一个好的 AI 算法影响有多大。

好了，回到我们的正题上来，继续谈谈视觉领域的应用。话说在学界发明并完善了 CNN 和 RNN 这两种网络后，针对网络灵活组合的应用大为增加。下面介绍两种典型的 CNN+RNN 合作典范。

4.4.2　看图说话：用文字描述一幅图像（BiRNN+CNN）

看图说话，顾名思义是让机器看一幅图，机器就能根据这幅图片所描述的物体或景象输出一句话。这个过程看似非常简单，对于人类来说这只是 4 岁孩童的智力水平！但机器要做成这样确实有难度，所幸依赖于 CNN+RNN 越来越融合的趋势，研究者们已经初步解决了这个问题，这也是我们要学习的第一个案例。

先来看看大致的方法：训练集给出图片和图片的几句描述性语句，比如图 4-55 的猫，再进行训练，最后对一张新的图片（测试图片）生成描述语句。来看看它用于评分的网络结构（如图 4-56 所示）。

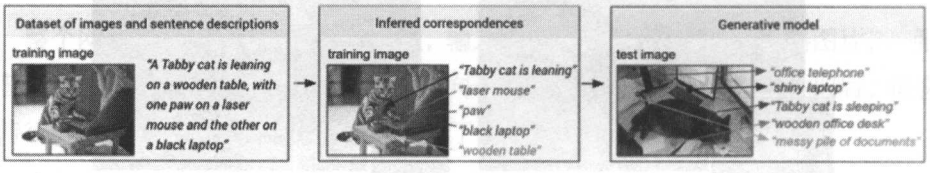


图 4-55　图像文字转化示意

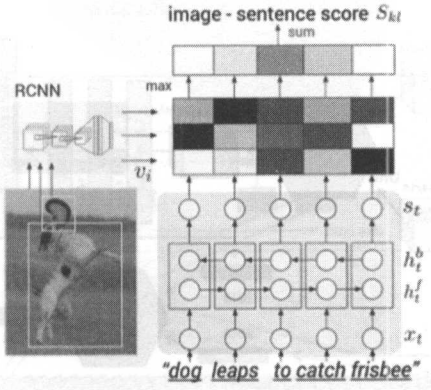


图 4-56 评分算法网络结构图

图 4-56 左侧是对一个 R-CNN 网络做目标区域识别，是否还记得 R-CNN（这篇论文出来的时候还没有 Fast R-CNN 及其后续版本），所以这里是用 R-CNN 做目标识别；图右侧是一个双向 RNN 网络，用于提取描述性语句信息。

将左边图像提取到的特征和 RNN 提取到的文本特征进行合并（内积），并输出打分。我们将训练好的网络稍微调整，就能用一张图片输出文字（如图 4-57 所示）。

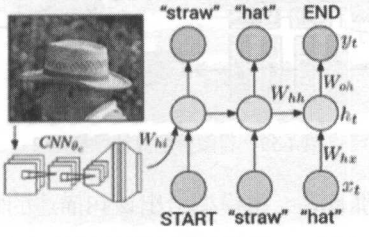


图 4-57 图像到文字的算法示意图

这个案例本质上是 CNN 叠加 RNN 网络。

第二个案例探讨关于 CNN+LSTM（RNN 的一种）变形结构的应用，也是用于处理图文。

比如看了一幅图片，并给机器一个有关这幅图片的问题，机器根据图片和问题，给出答案，这有点接近人类 5 岁孩童的智力了，比看图说话更进了一步，我们称为看图答题。算法结构如图 4-58 所示。

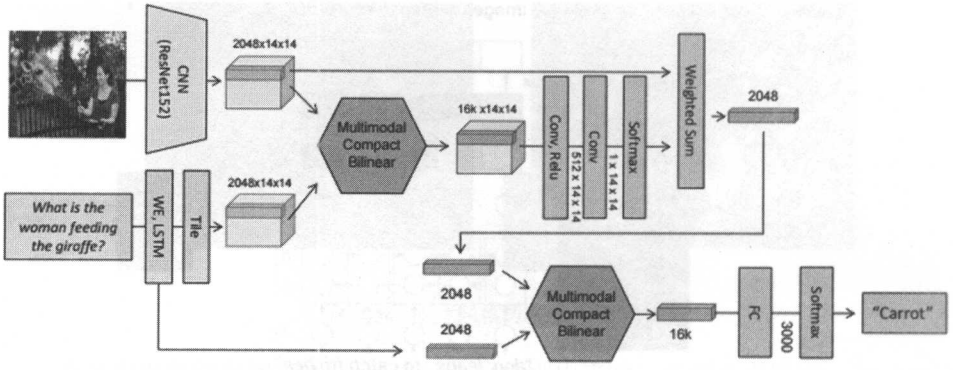


图 4-58　看图答题算法结构图 1

第三个案例是通过一句话的描述来定位识别图片中的物体（如图 4-59 所示）。

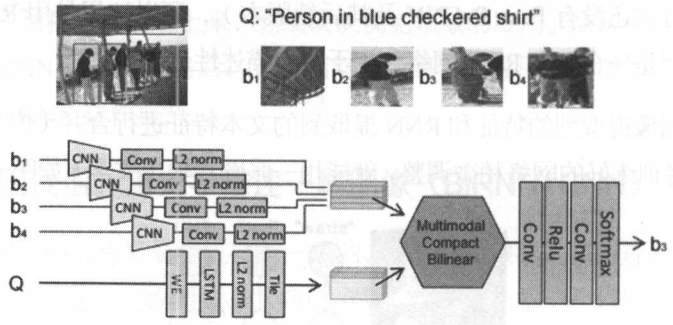


图 4-59　看图答题算法示意图 2

输入是一句描述和一张图片，由模型给出该句描述的特定对象，以上的例子最后输出是 b3，符合一句话的描述：“穿蓝色格子衬衫的人”。

CNN+RNN 的玩法很多，除了上面的文本和图片的混合玩法，还有人将 CNN+RNN 应用在语音识别中，并取得较好的效果，请看下文。

4.4.3　CNN 的有趣应用：语音识别

看到这个标题很奇怪，CNN 本来是用在图像上的，还能做语音应用吗？是的，用 CNN+RNN（GRU）方法能让语音识别的成功率上升到 99.24%，多种模型混合叠加后（类似随机森林）能达到 99.67%的识别精度（如图 4-60 所示）。

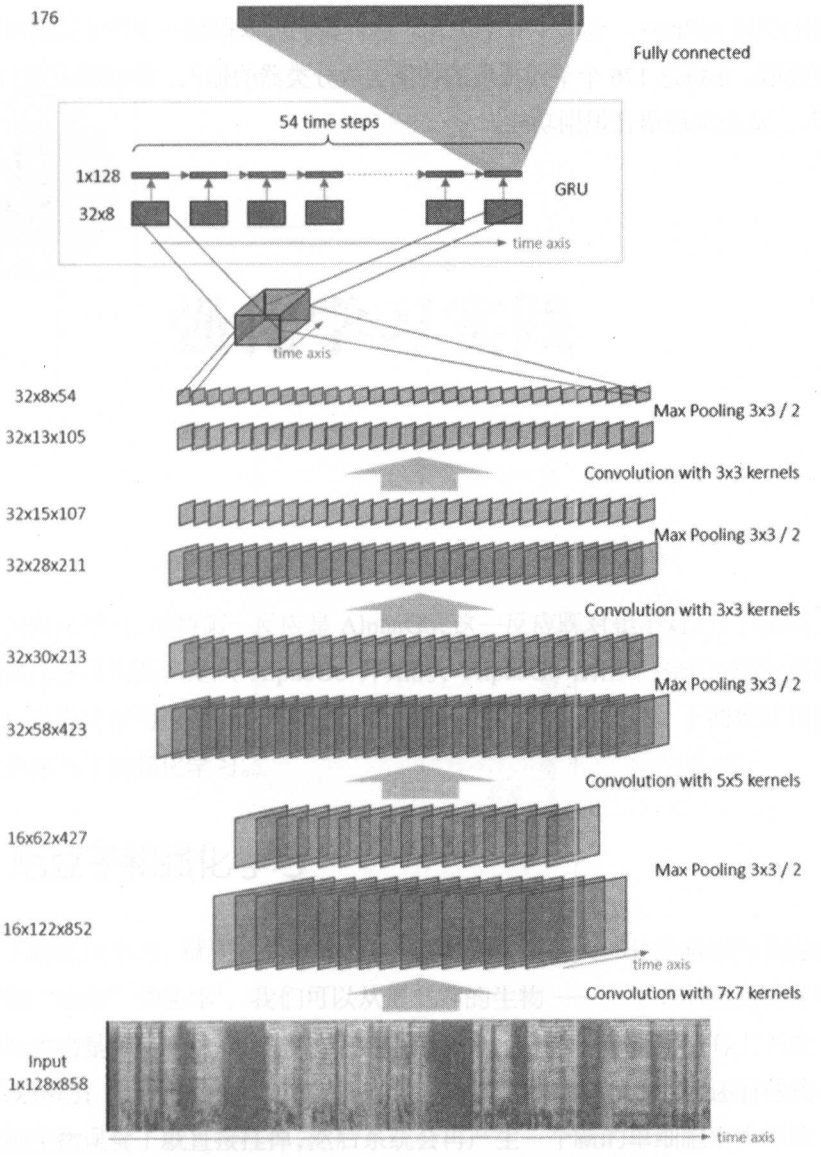


图 4-60 CNN+RNN 提高语音识别准确度

这个结构图中，从数据输入开始到第九层之前是个典型的 CNN 网络，而此结构将卷积层第九层之后的全连接层（FC）去掉，softmax 层去掉，加入一个 RNN 网络处理时序特征，将带有时序特征的语音输入数展开成一个二维网络，再进行处理并输

出，输出层是 softmax，包含 176 个单元。这样我们可以得到一个基于语音时间片段的特征提取。在将这 176 个单元提取的特征当成分类器的输入，就能够识别出基本的“音素”，从而实现语音识别功能。

5 强化学习实践

听到强化学习，英特第一反应是 AlphaGo，这一反应既对也不对。对于英特来说，接触到强化学习的概念是从 AlphaGo 开始的，AlphaGo 刚出来的时候确实震撼了一大批人，但强化学习这个概念其实在 AlphaGo 之前就早已出现。下面就让我们和英特一起来深入了解强化学习。

5.1 吃豆子和强化学习

要了解强化学习，就要从生物界找灵感，数据科学的大部分范畴都应该归结为实验科学和“空想”仿生学[1]，我们可以从最低等的生物——一个单细胞生物开始，看看单细胞生物是如何学习的。首先给单细胞生物设计一个场景，它只有上下左右四个方向可以移动；周围有微生物，单细胞生物可以吃，看能吃多少；但还有些病毒，如果单细胞生物误食了就直接挂掉，然后系统会再产生一个新的单细胞生物继续上面的循环，当然系统在 reset 这个单细胞生物时，已将之前遇到微生物（食物）和病毒（天敌）的经验输入到新的单细胞生物上。

从单细胞生物这种求生避险的本能，我们可以抽象出强化学习的以下几个概念。

1 作者个人的命名，不一定科学。

- 环境（environment）也就是边界或者说移动范围，还有一些规则，比如规定吃到东西单细胞生物就可以长大，吃到病毒就挂掉重新开始。
- 奖励（rewards）：这里的奖励有两个，一个是吃到微生物就可以成长，我们定义该奖励是正值；另一个是吃到病毒就减分或挂掉，我们定义该奖励为负值。
- 动作（actions）：也就是允许的单细胞生物的动作。

好，整个过程其实和吃豆子这个游戏很像，所以我们就以吃豆子为例（如图 5-1 所示）。

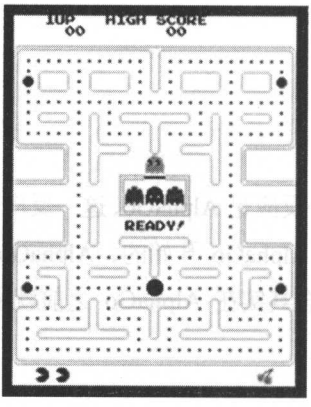


图 5-1　吃豆子游戏

这里有四处游荡的怪物，也有吃豆人（也就是我们的主角，吃了会加分的豆子）。而除此之外游戏的路径和图形就是环境。

我们按照之前单细胞生物的知识点再来梳理下。

- 状态（states）= 有的书上也称为 observation、environment。状态指的就是环境（environment）的状态，即在当前的情况下，每一次移动后，各个怪物的位置，以及豆子和整体环境的变化，一般情况下是游戏中任意时刻整个画面的一帧，将它作为输入状态（states）；
- 动作（actions）= 每个状态下，吃豆人什么样的动作；
- 奖励（rewards）= 每个状态时下，在动作（action）之后带来的正面或负面反馈，比如加分或扣分；

- 智能体（agent） = 这里指的是吃豆人。

将单细胞生物或者吃豆人这类最简单的和环境有交互的状态、动作等抽象出来后，我们希望继续深入了解单细胞生物或者吃豆人是如何从环境中学习到趋利避害的“本领”的，请继续往下看。

5.2 马尔科夫决策过程

吃豆人的游戏中，每一步动作后环境的状态会发生变化：可能吃了一个豆子，或者是往前走一步，或者被杀死，同时会带来正向奖励或负向奖励（负向奖励就是惩罚），沿着目前的这步变化，可以推导出后期的奖励。

换句话说，每一次动作后，都会对未来产生一个可能的路径，而我们的目标是在所有的路径中寻找最优的解。

举个例子，我们从上海乘车到北京，选择了最便宜的路线，此路线经过 10 个车站，第二站是南京：

上海→南京→……→北京

但如果除去始发点上海站，那么由第二站南京到最后的北京站：

南京→……→北京

这路线仍然是余下 9 个站之间最便宜的。

上海→南京的选择可以看成 action，选择了南京后，因为途径了 9 个站，我们理论上得到了多条路径，选择后发现还是原有的路径也就是上海→南京→北京这段路中间的南京→北京这段是最便宜的。接下来，我们再选择南京下一站徐州……这个过程一直重复下去，直至到达北京为止。

以上寻找最优路径的过程就是一个 MDP 过程（Markov decision process，MDP，马尔科夫决策过程），MDP 是在 MP（马尔科夫过程）上的一种优化，增加了一个关键元素：动作集，也就是前文提到的 actions（如图 5-2 所示）。

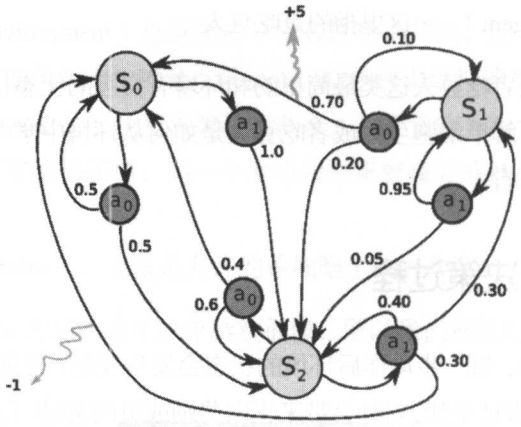


图 5-2　马尔科夫决策过程示意图

我们用图论来表示 MDP 的一个过程，MDP 过程表示为图中的状态转移，包括五个主要概念：

状态（state，S）：图中浅色的圆圈所示，为环境的“观测值”。

动作集（actions set）：图中深色的圆圈所示，在图中的状态转移过程中，动作集为｛a0, a1｝。动作就是 MDP 比 MP 增加的部分。

状态转移概率（Psa）：指定在状态 s 的情况下执行动作 a 后的状态转移概率。如在一个状态 s_0 下执行一个动作 a_0 后，将会有 0.5 的概率转移回 s_0，0.5 的概率转移到 s_2，但是我们有可能不知道状态的转移概率，即只有在观察到状态转移之后才知道下一个状态是什么。

奖励函数（reward）：作为强化学习中最重要的概念，奖励函数是与每个状态对应的，在智能体执行某个动作以后，环境过渡到下一个状态时给智能体的反馈，可以是正的或者负的，如图中浅色箭头所示，由于要求智能体执行某个动作后才能看到结果，所以称强化学习具有时间延时（time delay）性。假如我们在训练一条狗，当它做出正确的事情时，我们会说“乖狗狗”并给它吃的，而做错了一件事时，叫它“坏狗狗”，没有吃的，久而久之小狗就会知道怎样才是正确的做法。与此相同，正确地选择奖励函数，能够提高智能体的训练效果和训练速度。

折扣因子（discount factor，gamma）：是对未来奖励的不确定性的表示，状态转移的过程具有一定的随机性，时间间隔越久，未来的不确定性越强，未来能够获得奖励的可能性也就越小。所以选择一个［0，1］的一个值作为折扣因子，时间越是向后推迟，我们对未来的预测就要乘上更多的折扣因子。

说到这里，很多读者可能会想，这和最优路径选择好像差不多？这里有个误区，尤其是对于 AlphaGo 或者其他的游戏，强化学习和最优路径选择的区别是，强化学习过程不能够穷尽所有的路径，因为根本看不到最终的状态。比如，下象棋时我们可以穷尽所有的路径，因为象棋的棋盘相对较小。而对于围棋来说，我们没有可能计算完所有的情况，能算十步已经很了不起了，对机器来说一些简单的问题可以遍历，但围棋这样的要遍历一遍再找出最优路径是不可能的，所以强化学习不是找最优的，而是找对于目前状态来说可能是最优的，在这一过程中并没有模拟未来情况（这点也和最优路径有区别），而是直接计算出未来可能的分值。

说完了 MDP 的解释，再了解其基本公式，更深入地理解它是如何实现的。

公式 1：假设我们在描述这样一个转移状态，从 s_0 到 s_n，公式 1 描述了一条路径。

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \cdots$$

公式 2：我们的目标是使获得的总价值最大，总价值表示为公式 2，即从 s0 开始到一个 episode 结束获得的总价值，或者表述为未来的累计奖励值；R 表示**当前状态**下获得的奖励值。其中的 gamma 就是折扣因子，因为在状态转移的过程中未来的状态是随机的，也就是我们不知道未来会发生什么，所以对于未来的奖励乘上折扣因子，并且是随着时间呈指数增长。

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots .$$

公式 3：若用π表示一系列动作所组成的策略（policy）的话，那么总共获得价值就可以用一个价值函数$V^{\pi}(s)$来表示，即为在状态为 s=s_0，策略为π的条件下，价值的期望值（因为状态转移的过程具有随机性，所以这里的价值用期望来表示）。

$$V^{\pi}(s) = \mathrm{E}\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots \middle| s_0 = s, \pi\right]. \quad \pi : S \mapsto A$$

公式 4：对上式做简单的变形（将 gamma 提出来），若该式表示 s_0 开始到一个 episode 结束时的总价值，那么括号里面表示的就是从 s_1 开始到一个 episode 结束时的总价值，整理一下变为公式 5。

$$V^{\pi}(s_0) = E[R(s_0) + \gamma(\underbrace{R(s_1) + \gamma R(s_2) + \cdots}_{V^{\pi}(s_1)})|s_0 = s, \pi]$$

公式 5：$V^{\pi}(s)$表示状态 s 处对未来总价值的预测，后面的$V^{\pi}(s')$表示状态s'处对未来总价值的预测，因为下一时刻的状态由 action 和转移概率决定，所以有了这个公式，这个公式被称为“贝尔曼等式”，是处理 MDP 优化的重要概念。

$$V^{\pi}(s) = R(s) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s') V^{\pi}(s').$$

公式 6：而求解贝尔曼等式就相当于使函数最大化的过程，我们用 $V^{*}(s)$表示通过选择适当的策略获得最大的价值，这个过程可以从两个角度来考虑，值迭代（value iteration）和策略迭代（policy iteration）。

$$V^{*}(s) = \max_{\pi} V^{\pi}(s).$$

最后我们要感谢马尔科夫让我们在状态之间能游刃有余地进行切换：从状态 s 到状态s'我们不需要关心其他状态，只需要关心上一个状态即可，自然也就有了状态转移概率P(s'|s)一说。

5.3 理解 Q 网络

我们花了很多篇幅谈到马尔科夫决策过程和它的求解目标，那么这一切和强化学习又有什么关系呢？

从上面对 MDP 的了解来看，MDP 求解其实是一个估值过程，而这类的估值问题是强化学习里的一个关键目标：即每个状态之后都要选择一个动作，每个动作之后的状态也不一样，我们需要一个值来评估某个状态下不同动作的得分。如果知道这个得分，我们也就能选择一个使得分最优的动作来执行。这就是强化学习中的关键一步

动作估值（Action-Value function $Q^{\pi}(s,a)$）了，可以简单写作$Q(s,a)$，按照以上思路继而求最优动作估值（$Q^{*}(s,a)$），这个Q^{*}我们可以认为是最理想的值，只能无限逼近。为了达到最优的估值Q^{*}，需要不停地迭代，也就是每次根据新得到的 reward 和原来的$Q(s',a')$值来更新现在的 Q 值。我们来看看图 5-3 的结构图，这里的 DQN 指的是深度 Q 网络(Deep Q-Network)，大白话就是用和环境的交互来衡量或评价动作（action）的好坏。

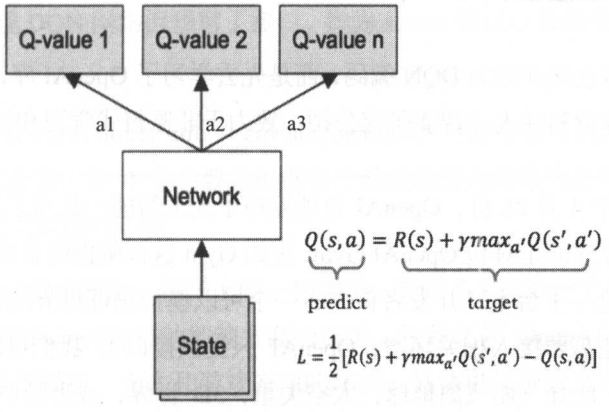


图 5-3 DQN 网络示意图

图 5-3 所示为 DQN 算法的网络示意图，中间的 Network 表示一个神经网络，输入为当前的状态，输出为对应不同的 action 的 Q 值（未来的总价值）。

这里给出 Q 版的贝尔曼等式，在训练 q-network 时，我们把等式右侧的值作为目标（target），使左侧的预测值能够接近这个目标。之所以这样选择是因为右侧比左侧多了一个 s 状态下所获得的奖励（reward）的信息，可以理解为右边比左边更有可信度。

Q 的学习过程：

```
def learnQ(self, state, action, reward, value):
oldv = self.q.get((state, action), None)
if oldv is None:
      self.q[(state, action)] = reward
else:
```

```
self.q[(state, action)] = oldv + self.alpha * (value - oldv)
```

请看最后一句，描述了旧有状态和目前输入状态下如何产生一个新的 Q 值。

以上，我们跟随英特深入理解了马尔科夫决策过程以及 Q 学习过程，这也是 DQN 网络的核心内容。英特现在要考虑实现一个 DQN 网络该怎么做？请看下节。

5.4 模拟物理世界：OpenAI

英特并没有立即开始为 DQN 编码，而是先去学习了 OpenAI 库，OpenAI 是什么呢？这是一家非营利性人工智能研究公司，致力于非监督式学习和强化学习的研究。

> 2016 年 4 月 28 日，OpenAI 对外发布了人工智能一款用于研发和比较强化学习算法的工具包 OpenAI Gym，正如 Gym 这词所指的意思（健身房）一样，在这一平台上，开发者首先有一个可以模拟物理世界的接口（要想真正模拟还需要接入相关环境，OpenAI 只提供接口），我们可以模拟各种物理环境，比如飞船飞离地球，太空失重，3D 世界，或者简单到一个 2D 游戏场景，比如吃豆子，另外开发者也可以把自己开发的 AI 算法拿出来训练和展示。

综上，OpenAI 是为强化学习创造一个虚拟的世界，这个世界模拟了现实生活的各种物理规律，或者就称为规律，在这里我们的智能体才能很好地完成交互，进而完成强化学习的过程。

OpenAI Gym 最重要的功能就是提供各种强化学习环境。下面代码中的这句话 env = gym.make('CartPole-v0') 是实例化一个 CartPole 环境。后面几句代码是在更新动画，并且在得到 action 后做一步环境的变化。

```
import gym
env = gym.make('CartPole-v0') #实例化一个 CartPole 环境
for i_episode in range(20):
    observation = env.reset()
    for t in range(100):
```

```
        env.render() #更新动画
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action) #推进
一步
        if done:
            break
```

其中最关键的是 action = env.action_space.sample()，Gym 里面直接用了一个随机采样，并没有实现 DQN 网络但预留了接口，作为 action 输出以及接受 reward 和 state 的反馈。

observation, reward, done, info = env.step(action)这一部分里每执行一步 action 都将有 4 个参数的反馈，这 4 个参数是根据每个游戏的游戏规则来反馈的。这里看出 Gym 确实是模拟了游戏的环境。

CartPole 环境要求平衡一辆车上的一根棍子，如图 5-4 的第一个环境表示。该图是 OpenAI Gym 提供的部分自动控制方面的环境。除此之外 OpenAI Gym 还提供了 3D 游戏、物理世界模拟、文本和游戏方面的环境，具体可以查看官方说明。

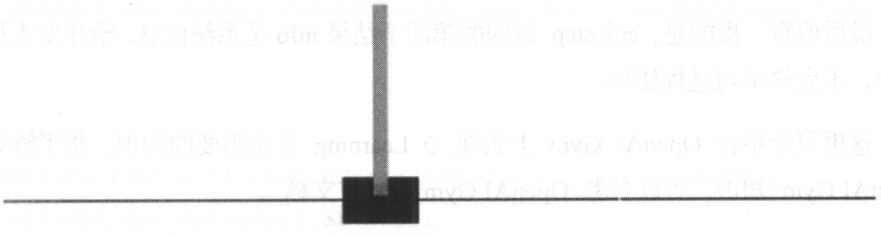


图 5-4 CartPole 游戏演示图

前文提到强化学习环境其实是马尔科夫决策过程，马尔科夫决策过程的四个基本元素分别是状态、动作、转移概率和奖励函数。对应 5.2 节提到的马尔科夫概念，我们做一个状态模拟，说说在 OpenAI 中如何实现表示马尔科夫决策过程的四个基本元素。

（1）状态：代码中的 observation 就是马尔科夫决策过程的状态。更正确地说是状态的特征。CartPole-v0 的状态特征是一维数组，比如 array([-0.01377819, -0.01291427, 0.02268009, -0.0380999])。有些环境提供的状态特征是二维数组，比如 AirRaid-ram-v0 环境提供的是二维数组表示的游戏画面。

observation = env.reset() 是初始化环境，设置一个随机或者固定的初始状态。env.step(a1) 是环境接受动作 a1，返回的第一个结果是接受动作 a1 之后的状态特征。

（2）动作：代码中的 action 就是马尔科夫决策过程中的动作。CartPole-v0 的动作是离散型特征。在 OpenAI Gym 中，离散型动作是用从 0 开始的整数集合表示，比如 CartPole-v0 的动作有 0 和 1。另一种动作是连续型，用实数表示。

（3）奖励函数：在 OpenAI Gym 中，它提供给强化学习一个环境。step 函数就是你的手柄，在手柄上按键相当于传入的值。

代码中的 reward 就是马尔科夫决策过程中的奖励，用实数表示。在 OpenAI Gym 中，奖励函数也没有显式地表示出来，也是通过 env.step(a1)的结果表示。env.step(a1) 返回的 reward 满足奖励函数。

（4）转移概率：指的是玩游戏时当前画面执行了一个操作后，所呈现下一帧（下一帧有无穷种可能）画面的可能性，在 OpenAI 中无法显式地观测到转移概率。

最后值得一提的是，env.step 返回的第四个结果 info 是系统信息，给开发人员调试用，不允许学习过程使用。

这里只介绍在 OpenAI Gym 上实现 Q Learning 算法需要的知识。想了解更多 OpenAI Gym 知识，可以参考 OpenAI Gym 的官方文档 。

5.5 实现一个 DQN

5.5.1 DQN 代码实现[1]

学习了 OpenAI 的使用后，英特才真正开始着手实验 DQN 代码。

1 代码参考自 https://github.com/devsisters/DQN-tensorflow。

DQN 代码的安装和训练非常简单，它是依赖 OpenAI Gym 的，所以首先必须装一个 Gym，请参见下面这行安装脚本：

```
$ pip install tqdm gym[all]
```

接着为 Breakout 这个游戏训练一个模型：

```
$ python main.py --env_name=Breakout-v0 --is_train=True
$ python main.py --env_name=Breakout-v0 --is_train=True
--display=True
```

接下来可以测试和记录游戏的状态：

```
$ python main.py --is_train=False
$ python main.py --is_train=False --display=True
```

前文在提到 OpenAI 时，讲到了 OpenAI 预留接口，并没有实现 DQN。为了更深入地理解 DQN 的结构，我们了解下 DQN 的代码实现，先从 main 函数入手，追踪下 Agent 函数。

```
  def main(_):
    gpu_options = tf.GPUOptions(

per_process_gpu_memory_fraction=calc_gpu_fraction(FLAGS.gpu_fracti
on))

    with
tf.Session(config=tf.ConfigProto(log_device_placement=True,
gpu_options=gpu_options)) as sess:
    # with
tf.Session(config=tf.ConfigProto(log_device_placement=True)) as
sess:
      config = get_config(FLAGS) or FLAGS

      if config.env_type == 'simple':
        env = SimpleGymEnvironment(config)
```

```
    else:
      env = GymEnvironment(config)

    if not FLAGS.use_gpu:
      config.cnn_format = 'NHWC'

    agent = Agent(config, env, sess)

    if FLAGS.is_train:
      agent.train()
    else:
      agent.play()

if __name__ == '__main__':
  tf.app.run()
```

关键的一句在这里：agent = Agent(config, env, sess)，这是 agent 处理动作和状态的主要函数，这句话将往 Agent 函数里传送 config（配置文件），env（环境），sess（TF 的对象）。

接下去看看 Agent.py 里有哪些关键点？在 agent 文件的 play 函数中，我们找到了以下几行如何处理动作和状态的关键点：

```
# 1. predict
action = self.predict(self.history.get())
# 2. act
screen, reward, terminal = self.env.act(action, is_training=True)
# 3. observe
self.observe(screen, reward, action, terminal)
```

这些关键点的含义如下。

第一步用历史数据预测下一个 action。

第二步把预测出来的这个 action 放入环境中，得到下一个 reward。

第三步在新的 action、reward 基础下，继续‘看’整体的状态（state）。

在 train 函数中，同样能找到它们。

```
# 1. predict
action = self.predict(self.history.get())
# 2. act
screen, reward, terminal = self.env.act(action, is_training=True)
# 3. observe
self.observe(screen, reward, action, terminal)
```

除此之外，还有以下和 play 函数不一样的地方。

```
if terminal:
  screen, reward, action, terminal = self.env.new_random_game()

  num_game += 1
  ep_rewards.append(ep_reward)    # 存储每个 episode 的累积 reward
  ep_reward = 0.
else:
  ep_reward += reward

actions.append(action)
total_reward += reward

# 每 self.learn_start 次统计训练结果,并在适当时候保存模型
if self.step >= self.learn_start:
  if self.step % self.test_step == self.test_step - 1:
    avg_reward = total_reward / self.test_step      # 平均每个 step
的 reward
    avg_loss = self.total_loss / self.update_count  # 每次参数更新
的平均 loss
    avg_q = self.total_q / self.update_count        # 每次参数更新
的平均 q 值
```

```
        try:
            max_ep_reward = np.max(ep_rewards)          # 曾经到达的最高
的得分
            min_ep_reward = np.min(ep_rewards)          # 曾经到达的最低
的得分
            avg_ep_reward = np.mean(ep_rewards)         # 平均得分
```

以上就是 Agent 的运作方式，接着我们回到 DQN 函数中的 def build_dqn(self)，发现函数主要是构建 dqn graph 结构，包括了 train-network、target-network、pred_to_target、optimizer、summary。

其中 train-network、target-network 都是差不多的卷积神经网络，用于对 state 的观测。

在 config.py 中主要是一些配置信息，以下是 M1 配置信息，环境的类型设置为 'detail' action_repeat=1。

```
class M1(DQNConfig):
    """
    某种 训练方式 的配置,其中:
    AgentConfig 为 DQN 的配置
    EnvironmentConfig 为 gym environment 的配置

    """
    backend = 'tf'
    env_type = 'detail'
    action_repeat = 1
```

结合论文[1]我们试着修改一些参数，得到 M4 这个配置。

```
class M4(DQNConfig):
    backend = 'tf'
    env_type = 'simple'
```

1 Mnih, Volodymyr, et al. "*Human-level control through deep reinforcement learning.*" Nature 518.7540 (2015): 529-533.

```
action_repeat = 4
```

以上 M4 配置环境的类型设置为'simple'，而 action_repeat 设置为 4，表示每一 action 在训练内重复 4 次，同时让 reward 持续累加，个人认为这样能够起到加快训练的目的。

我们用 M1 和 M4 的配置跑出的模型分别去玩 BreakOut 游戏。用 M1 配置训练了 BreakOut 游戏 2000 万次，得到模型后，再用这个模型去玩游戏，游戏在玩到只剩一个人的时候，得到 32 分，而这时 DQN 网络预测的最高得分是 55 分（如图 5-5 所示）。

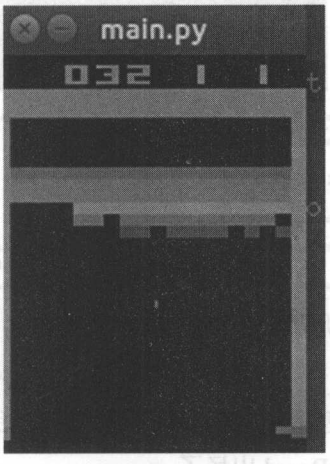


图 5-5 M1 模型打砖块 breakout 游戏演示图

```
[39] Best reward : 55
================================
  3%|▌                    | 309/10000 [00:10<05:23,
29.97it/s]
```

再来看 M4 模型，在训练游戏 1600 万次之后、M4 模型操作将游戏玩到只剩 3 个人时，已经得到 35 分，而 DQN 网络预测的最高得分是 209（如图 5-6 所示）。

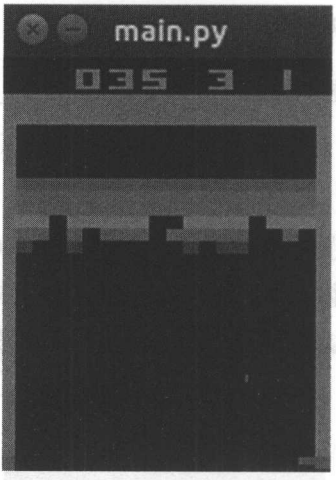


图 5-6 M4 模型打砖块 breakout 游戏演示图

```
[12] Best reward : 209
==============================
19%|                    | 1888/10000 [01:03<04:30, 29.95it/s]
```

从这个过程中，我们能明显看出 M4 的参数优化得比 M1 要好，但有读者要问了，如果我还想观察中间过程怎么办？我甚至想看每一个 reward，还有模型训练中的准确率和最终得分，等等，这该怎么办？有没有更好的方法能够将整个训练过程图表化出来呢？那下面就要请 TensorBoard 出场了。

5.5.2 DQN 过程的图表化

TensorBoard 是 TensorFlow 中的可视化工具， TensorFlow 的版本还在不停地迭代，所以 TensorBoard 的具体使用步骤请直接参考官方网站。这里只尝试用 TensorBoard 将训练过程中的对应日志生成相应图表，为了观察对比，我们将一些关键指标罗列出来。

（1） 在所有的标量图（单值的变化）中横轴表示训练的 step 数，纵轴表示每个量的数值，反映出在训练过程中我们关注的每个量的变化情况。

图 5-7 为 m1 模型的结果中的标量变化曲线，图中各参数含义如下：

‘average.reward’：平均每个 step 的 reward。

'average.loss'：每次参数更新的平均 loss。

'average.q'：每次参数更新的平均 q 值。

'episode.max reward'：曾经到达的最高得分。

'episode.min reward'：曾经到达的最低得分。

'episode.avg reward'：平均得分。

'episode.num of game'：游戏名字，如：breakout。

'training.learning_rate'：学习率。

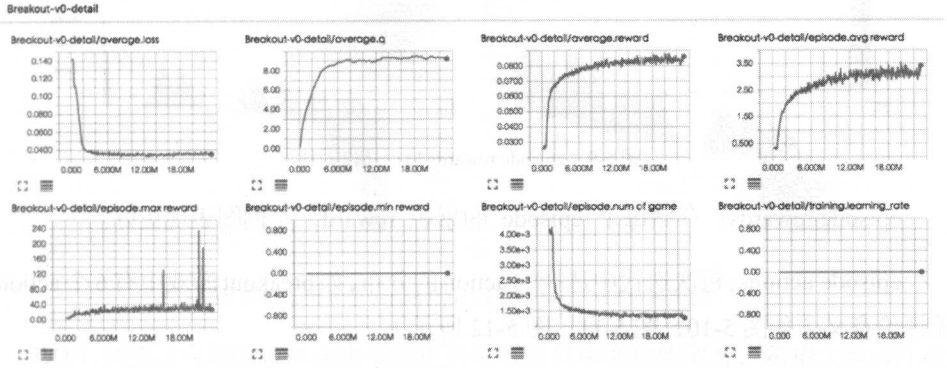


图 5-7 标量变化曲线

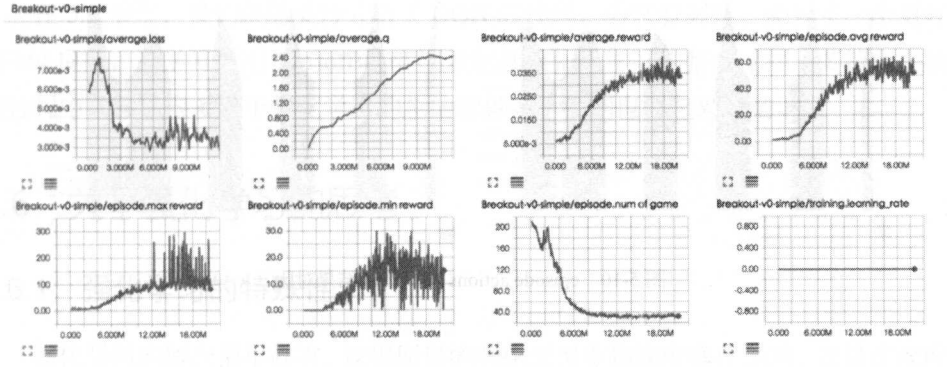


图 5-8 M4 模型标量变化曲线

（2）在所有的矢量图（多值的变化）中：黄颜色（左侧）代表 M1 模型的结果，

绿颜色（右侧）代表 M2 模型的结果。

TensorBoard 的矢量图包括三个维度，右侧的坐标为 step 数（即表示时间），下侧的坐标表示 Tensor 中的索引，每一个横轴上的图形表示在当前时刻（step）不同的值上面的概率分布（如图 5-9 所示）。

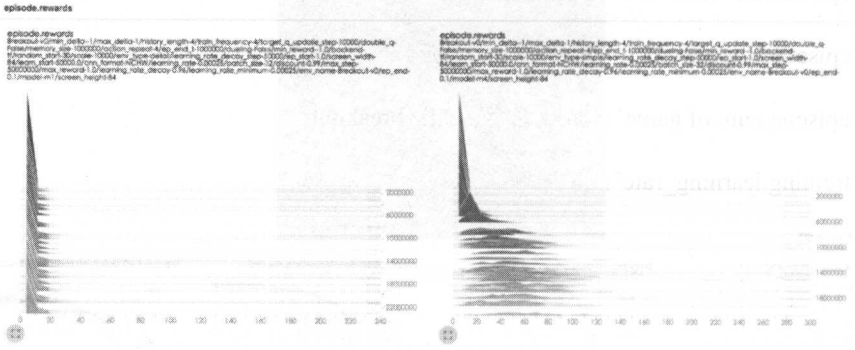


图 5-9　episode rewards M1、M2 对比

episode rewards：存储每个 episode 的累积 reward，左侧图表现更好。

episode actions：每次迭代中选择的 action 的分布，如 breakout 游戏中有 6 个 actions，无评价意义（如图 5-10、图 5-11、图 5-12 所示）。

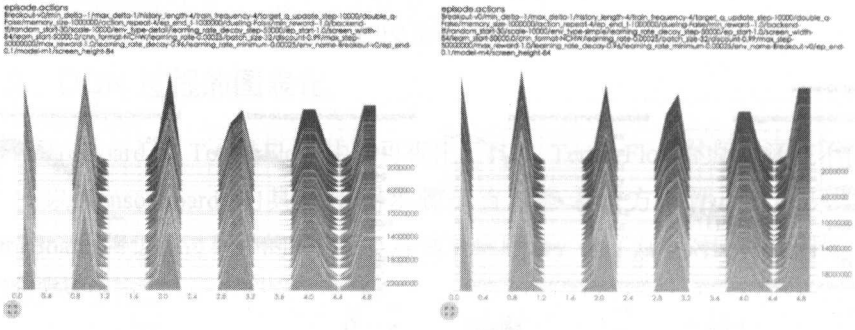


图 5-10　episode actions M1、M2 结果对比

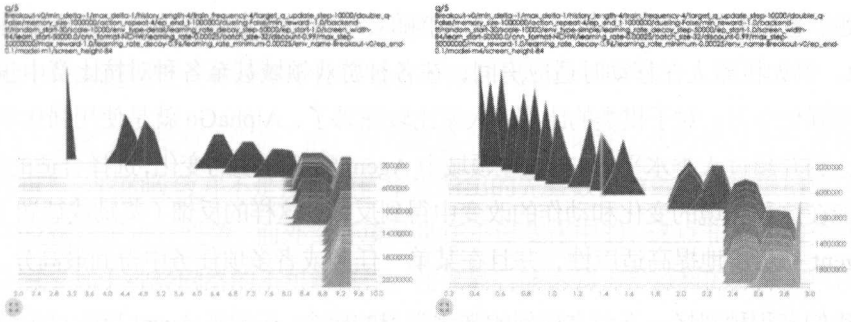


图 5-11 第 6 个 action 的 q 值变化 M1、M2 结果对比

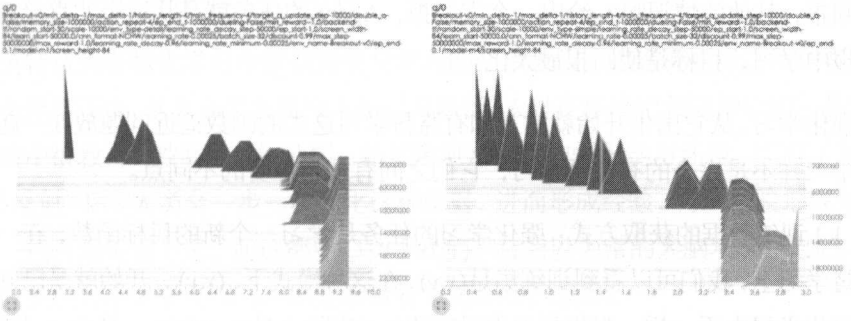


图 5-12 第 0 个 action 的 q 值变化 M1、M2 结果对比

DQN 训练过程用图表展现出来能更好地观察其中变化的情况，希望大家善用此工具。

在 5.5 节里，我们跟随英特分析了 DQN 的代码，尝试着调参，得到了一个更好的 M4 模型，又学习了用 TensorBoard 图表化展示强化学习的整个过程。英特在实践的过程中又多了很多对于强化学习的总结和思考，我们也一起来总结下。

5.6 关于强化学习的思考

5.6.1 强化学习的特殊性

强化学习的概念很早就有，这里所说的强化学习专指深度强化学习，这是在深度网络发展起来后，重新利用强化学习的机制而发明的算法。

强化学习当然不能解决所有的问题，但我们要了解强化学习究竟能够解决哪一类

的问题。通常来说，强化学习适合那些能够随着环境的改变而强化自己的智能体——Agent。例如机器人在移动时适应房间，在各种游戏领域甚至各种对抗比赛中也都可以应用强化学习；对于棋类的比赛，大家比较熟悉了，AlphaGo 就是使用强化学习的方法达到并超过人类水平的。在这些领域中 Agent 随着环境的变化，选择合适的动作，并且能在这种环境的变化和动作的改变中得到反馈，这样的反馈（奖励或惩罚），将使 Agent 一步步地提高适应性，并且在某单一任务或者多项任务中得到最高分。

我们来稍稍回忆一下前文提到的强化学习的概念。在训练 Agent 玩吃豆子游戏时，“环境”可以在这一局游戏结束并且胜利时给出一个正回报，而在游戏失败时给出一个负回报，其他的情况呢？给出一个零回报，Agent 的任务就是从这个非直接有延迟的回报中学习，目标是使回报最大化。

强化学习，从它出生开始就常常和有监督学习这类的函数逼近问题放在一起讨论，强化学习并不是完全的有监督学习，它们之间有几个重要的不同点。

（1）训练数据的获取方式：强化学习的任务是学习一个新的目标函数，在一般的有监督学习上，我们可以看到训练集是(x,y)，在这种模式下，(x,y)一开始就是已知的，而在强化学习中不一样。强化学习中对应的(x,y)实际上是$<s, \pi(s)>$，其中 s 是状态，而$\pi(s)$是在这种状态下的最优化动作。强化学习中训练信息是从跟环境中互动而得到的值。而环境也不是一成不变的，Agent 新一轮的动作，又对环境产生影响。

（2）探索：在强化学习中，Agent 通过其选择的动作序列影响训练样例的分别。这产生了一个问题：哪种实验策略可产生最有效的学习？学习器面临的是一个权衡过程，是选择探索未知的状态和动作（收集新信息），还是选择利用它已经学习过的、会产生高回报的状态和动作（加强神经元之间的联系）。

（3）目标不同：强化学习的数据是序列的、交互的、并且还有反馈的（reward）。这就导致了它与监督学习在优化目标的表现形式的根本差异：强化学习本质可以算作是一个决策模型，监督学习更偏向于在固定数据中寻找答案。强化学习是 Agent 自己去学习，监督学习是跟着设计者给出的既定方向在收敛。

（4）长期学习：不像一般的函数逼近任务，类似一个机器人学习问题，经常要求此机器人在相同的环境下使用相同的传感器学习多个相关任务。怎样捡起一个球以及

怎样从打印机中取得打印纸等。这就要求算法使用先前获得的经验或知识在学习新任务时减小样本复杂度，并且整个过程是在不停迭代的。

（5）强化学习是有“生命”的：你可能也不知道你训练出来的模型到底能做到什么“程度”，它的收敛并不是按照我们事先给出的先验数据收敛，而是根据整个外部环境的反馈数据进行收敛，而外部环境又是在不停变化中的，这点非常像动物或人的学习方式；加上神经网络的不可解释性，往往最后能达到出乎意料的效果，可以说强化是有“生命”的，或者说是有自我进化能力的。

5.6.2 知识的形成要素：记忆

英特认为在强化学习中要找到终身学习的开启大门，或者说要将这种“有生命”的状态延续下去，就有必要去模拟人的学习路径。人是在环境中获取知识的一种动物，在环境中被教育，被奖励，被惩罚，非常像前文模拟的强化学习环境，而有了基础的“神经反射”后，人类会一步一步强化这种反射，进而形成经验，再从经验形成知识，最后一代代地传递下来。而这都要归功于我们一直习以为常的大脑功能：记忆。所以这部分我们先不谈强化学习，而是来看看记忆网络的发展。

提到具备记忆的网络我们的第一反应基本都是 LSTM。它巧妙地利用了几个门（gate）实现了对是否记忆的控制。我们认为这种模式实际上是不完全的知识存储，因为 LSTM 只会对一个序列（sequence）里的目标进行记忆，而且严格地一次只输出一个字符或一个词（如图 5-13 所示）。

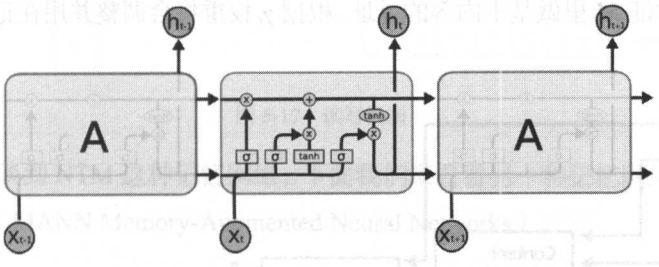


图 5-13 长短时模型（Long Short Term Model，LSTM）通过隐性的共享记忆结构，不完全地实现知识的存储。

LSTM 的出现起码解决了神经网络能否记忆的问题，虽然解决得还不够好。图 5-14 中的这位就是专门为了解决记忆问题而产生的神经图灵机（Neural Turing

Machines，NTM）。

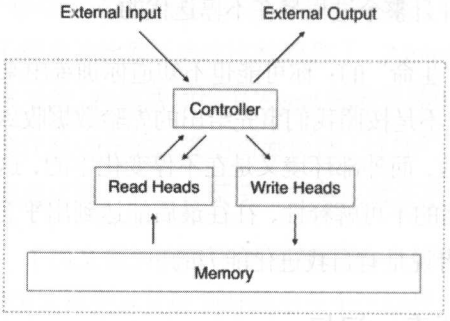


图 5-14　神经图灵机 NTM

2014 年 DeepMind 的同学们提出一种结构：神经图灵机，大家都知道图灵机是对外部环境的刺激做出的一种反馈（想想强化学习），而神经图灵机参考了这种反馈。

神经图灵机（NTM）架构包含两个基本组件：神经网络控制器和存储器组。图 5-14 提供了 NTM 架构图。像大多数神经网络一样，控制器通过输入和输出向量与外部世界交互。与标准网络不同，它还使用选择性读写操作与存储器矩阵（Memory）交互。类似于图灵机，我们将参数化这些操作的网络输出称为“Heads”（包含读写）。

这个结构可以让 NTM 在只接受少量的新任务观测情况下或者说在外部的刺激下实现快速学习记忆。而这个简单的结构，据说是模拟大脑皮质的记忆功能。

接着来看看 NTM 寻址问题（如图 5-15 所示），关键向量 $\boldsymbol{k}_t$，关键的强度 $\boldsymbol{\beta}_t$ 被用来在存储举证 $\boldsymbol{M}_t$ 里做基于内容的寻址。根据 $\boldsymbol{\gamma}_t$ 权重将会调整并用在记忆的访问上。

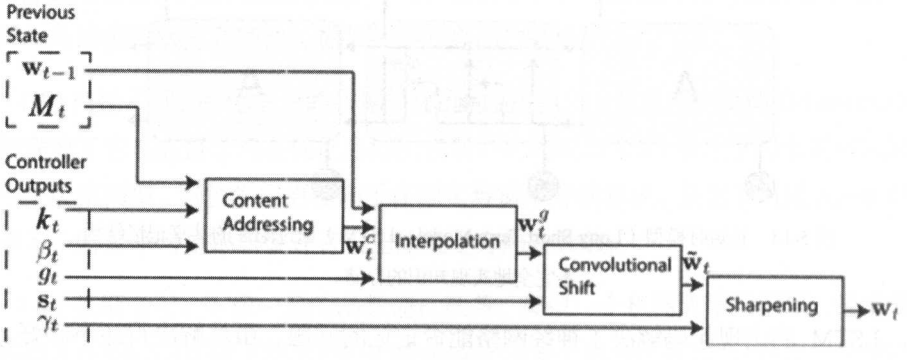


图 5-15　寻址机制的流程图

存储记忆的模块确定了、寻址机制确定了，但是如何往里进行读或者写呢？首先要让读写具有可区分性，以便让神经网络学习读取和写入的位置。这比较麻烦，因为内存地址是完全离散的。NTM 采取一个非常聪明的解决方案：每一步都进行读写（如图 5-16 所示），只是在不同的范围内进行，图 5-17 描述了读取写入的具体流程，其中图 5-17 的左边为读取，右边为写入。

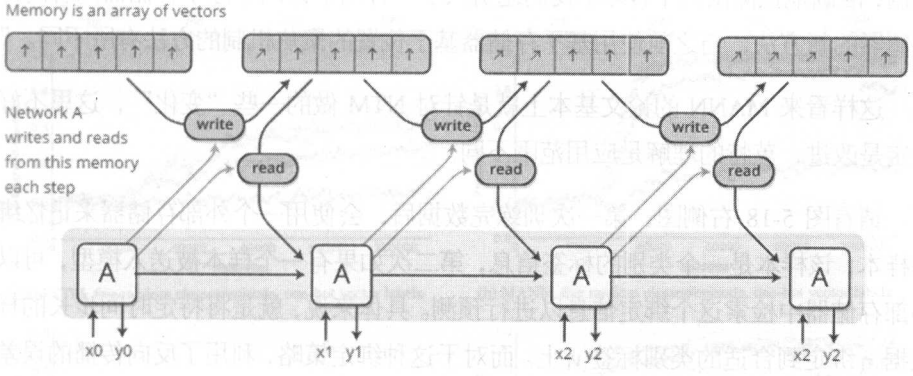


图 5-16 每一步都有读写[1]

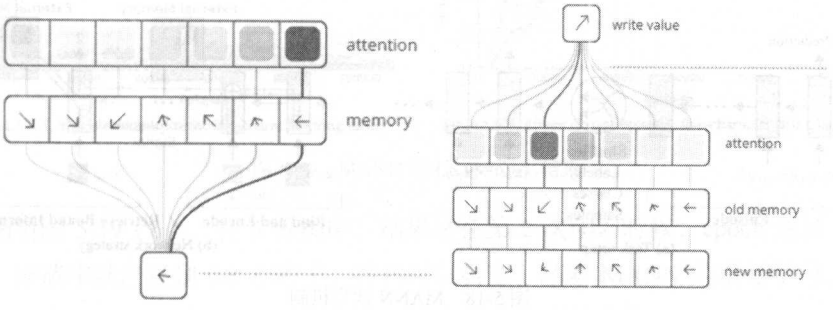


图 5-17 读写机制

以上我们谈到 NTM 这种记忆网络，下面我们来看看另一种记忆网络——记忆的进阶 MANN[2]（MANN Memory-Augmented Neural Networks）。

MANN 论文的摘要这样写道：“尽管最近在深层神经网络的应用方面取得了突

1 参考自 http://distill.pub/2016/augmented-rnns/。

2 参考自论文 *One-shot Learning with Memory-Augmented Neural Networks*。

破，但是一个持续挑战是“一次性学习”。传统的基于梯度的网络需要大量的数据来学习，常通过广泛的迭代训练。当遇到新数据时，模型必须无效地重新学习它们的参数，以充分地将新的训练集信息合入模型。而具有增强记忆结构的算法，诸如神经图灵机（NTM），能提供快速编码和检索新信息的能力，因此可部分地消除常规模型的缺点。在这里，我们演示了记忆增强神经网络快速吸收新数据的能力，并利用这些数据，准确地预测出几个样本。我们还引入了一种用于访问专注于存储器内容的外部存储器的新方法，与之前使用基于存储器基于位置的聚焦机制的方法有所不同。”

这样看来 MANN 的论文基本上就是针对 NTM 做的一些“变化”，这里不好说一定是改进，英特的理解是应用范围不同。

请看图 5-18 右侧图，第一次训练完数据后，会使用一个外部存储器来记忆绑定的样本，该样本是一个类别的标签信息，第二次如果有一个样本被送入模型，可以从外部存储器中检索这个绑定信息以进行预测。具体来说，就是将特定时间步长的样本数据 x_t 绑定到合适的类别标签 y_t 上。而对于这种绑定策略，利用了反向传播的误差信号，用该信号对时间较早的权重更新调整，也就是更新了绑定策略。

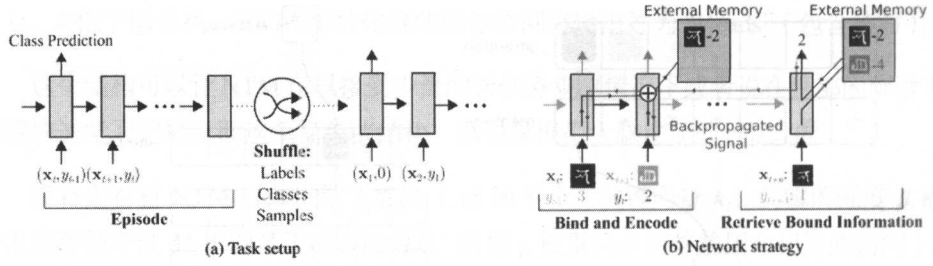


图 5-18　MANN 读写机制

以上的这种信息被传递进模型，被整理计算后决定是存储下来还是丢掉，这种机制（任何让我们有选择地把信息从模型的一个地方引导到另一个地方的机制）可以被认为是一种注意力机制。分类模型里的属于“读”注意力，生成模型里的属于“写”注意力或“生成”注意力，对输出变量选择性地更新。不同的注意力机制可以用相同的计算工具实现。

综上，有别于 NTM 由信息内容和存储位置共同决定存储器读写，MANN 的每次读写操作会选择最近利用的存储位置，这种选择是和时间相关的，因此读写策略完全

由信息内容和绑定时间所决定。MANN 实现了知识的高效的归纳转移（Inductive transfer）——新知识被灵活地存储访问，基于新知识和“长期”经验对数据做出精确的推断。

研究者使用实验数据集 MNIST 和 Omniglot，为了节省训练时间，图片被规则化成 20×20 大小，对比算法为 LSTM（如图 5-19 和表 5-1 所示）。

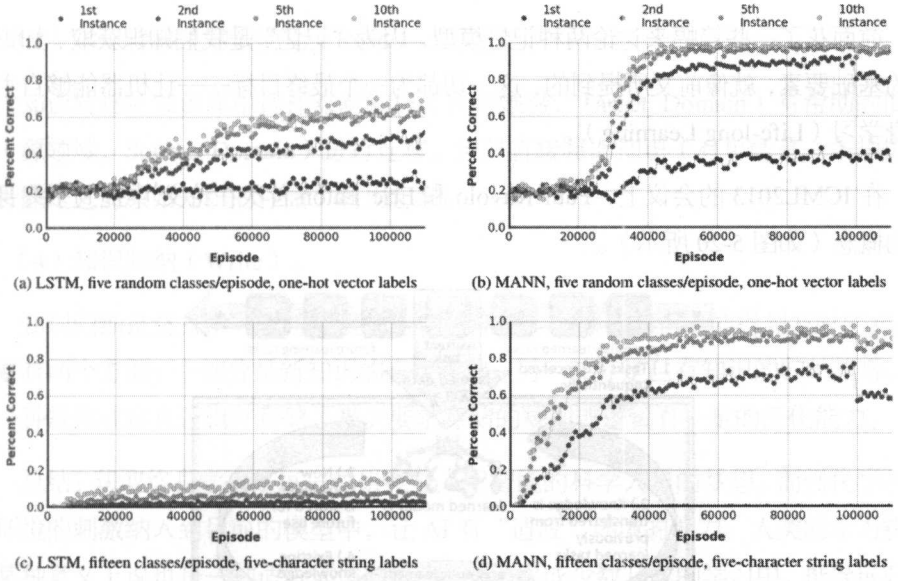


图 5-19 MANN 与 LSTM 对比结果

请看图 5-19（b），在 5 分类中，结果是在第 2 次学习时候，epoch 达到 40000 次后，准确率就已经到 0.8 左右了。并且可以看出，随着 MANN 的重复学习，学习知识的速度有一个陡峭的上升。这跟人类的学习曲线有些类似。

表 5-1 MANN 与 LSTM 对比结果表

MODEL	INSTANCE (% CORRECT)					
	1ST	2ND	3RD	4TH	5TH	10TH
HUMAN	34.5	57.3	70.1	71.8	81.4	92.4
FEEDFORWARD	24.4	19.6	21.1	19.9	22.8	19.5
LSTM	24.4	49.5	55.3	61.0	63.6	62.5
MANN	**36.4**	**82.8**	**91.0**	**92.6**	**94.9**	**98.1**

本节到这里就结束了，我们做一个强化学习和 NTM/MANN 的总结。

强化学习：强调如何基于环境而行动，以取得最大化的预期利益。也可以说是*Agent*不停适应环境的能力，或者可以说是形成“条件反射”的一种方式。

NTM/MANN：从学到的知识中记忆、归纳，形成规则应用到其他事物的判断中的能力。当然目前只能做到记忆并没有太多的归纳或推导能力，但未来可以尽情想象。

5.6.3 终极理想：终身学习

前面花了一些篇幅来讨论两种记忆模型，因为“记忆”是我们知识获取、知识形成的基础要素，就像前文所提到的，这一切都为一个最终目标——让机器能够自主地终身学习（Life-long Learning）。

在ICML2013的会议上，Paul Ruvolo和Eric Eaton首次在论文[1]中提到了终身学习的概念（如图5-20所示）。

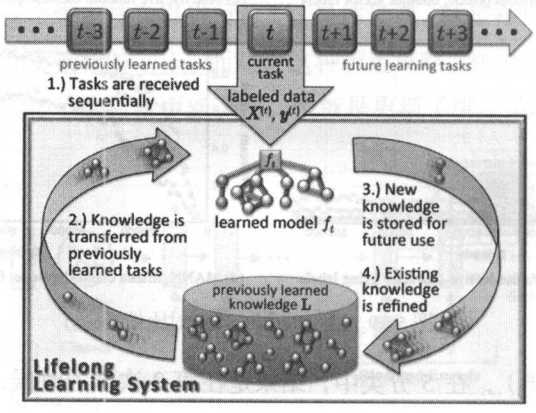


图5-20 终身学习示意[2]

英特在吸收论文中终身学习的概念后，认为一个终身学习系统应包含以下四个基础部分。

（1）知识库（Memory）。

知识库是图中下方的圆柱体，这里存储了前期学到的知识。它由输入和输出构成，

1 *An Efficient Lifelong Learning Algorithm*。

2 引用自 ELLA: *An Efficient Lifelong Learning Algorithm*。

输入的一部分是新知识直接存储，另一部分是已有知识的归纳推导，形成一种规则；而输出则是知识的应用或传递。

（2）控制器（Controller）。

控制器对 Read 和 Write 直接进行操作，并考虑到学习的顺序和指向性；而顺序、指向性、以及随着时间推移变化的知识存储将对泛化能力与学习代价产生影响。

（3）知识转移或知识使用（Read）。

知识转移从知识库中选择对新知识（目标领域，Target Domain）有帮助的旧知识（源领域，Source Domain）进行迁移。或是直接将旧知识（有可能是一条规则，有可能是之前原始知识的存储）利用起来。

（4）知识归纳（Write）。

知识归纳是终身学习系统中至关重要的环节，以保证知识库能得到及时的更新。主要有两个方面：一部分是新知识的直接存储，另一部分是已有知识的归纳推导，形成一种规则或自有逻辑并存储下来。要求存储的规则或逻辑有一定的泛化能力。

总结：获取终身学习是每一位研究强人工智能的科学人员的梦想。而强化学习是将环境的刺激纳入到目前的模型中，让 AI 有“适应”环境的能力。人类的学习机制从某种意义上说也是一种适应环境的行为：从一次或少数几次的学习中，推导或归纳出一个规律，并将这个规律应用到新的判断中，这种能力将可能帮我们打开强人工智能的大门。

6 预测与推荐

6.1 从 Google 的感冒预测说起

埃博拉病毒（Ebola）最初于几内亚被发现，随后扩展至香港、赖比瑞亚及奈及利亚。此次疫情是 1976 年发现埃博拉病毒以来最严重的疫情。世界卫生组织（WTO）公布至 2014 年 10 月 14 日止已有 8997 个病例，已经夺去了 4493 条人命。此次疫情已被正式列为“国际间关注的公共卫生紧急事件”，类似的紧急事件在过去仅发布两次，2009 年的 H1N1 新型流感疫情正是其中之一。

我们或许可以从过往的案例开始了解大数据对于疾病预测的应用。Google 于 2008 年推出 Google 流感预测趋势（Google Flu Trends，简称 GFT）。那么 GFT 想干什么呢？GFT 认为在流感爆发的季节，人们用像 Google 这样的搜寻工具，搜寻对应流感的相关措施与资讯的比例将会增加，通过分析无数笔匿名的流感关键字数据，像“咳嗽”、“发烧”和“疼痛”这样的字眼，就可以准确且快速地预测流感将在哪里出现以及其散布的范围，所以说白了这是一个基于大数据的词频分析工具，起码一开始是这样的。那这个工具到底有没有用？我们来看下面的曲线（如图 6-1 所示）。

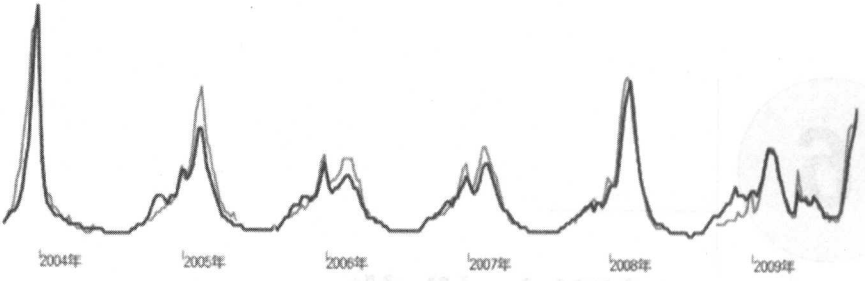


图 6-1　美国流感估测病例数 2004-2009

美国流感估测病例数，深色为 Google 流感预测趋势所得之曲线，浅色为美国联邦疾病控制与预防中心所获得之曲线。

根据以往的资料分析，Google 流感预测趋势的确取得了与美国联邦疾病控制与预防中心十分相似的曲线，而且还更即时。Google Trends 曾经有过多次对流感的成功预测，包括 2011/12 年的美国流感、2007/08 年瑞士流感、2005/06 年德国流感、2007/08 年比利时流感等。

Google 的流感预测从表面上看是成功的，一直到 2012 年都有很完美的数据，大家都将该案例当成教科书式的大数据案例，来证明大数据的有效性。

但是在 2012 年至 2013 年的流感流行季节里，GFT 过高地估计了流感疫情；在 2011 年至 2012 年则有超过一半的时间过高地估计了流感疫情。从 2011 年 8 月 21 日到 2013 年 9 月 1 日，GFT 在为期 108 周的时间里有 100 周的预测结果都偏高。图 6-2 中的上图为对流感样病例门诊数的预测结果；图 6-2 中的下图是几大预测机构相对真实数据的偏差值。

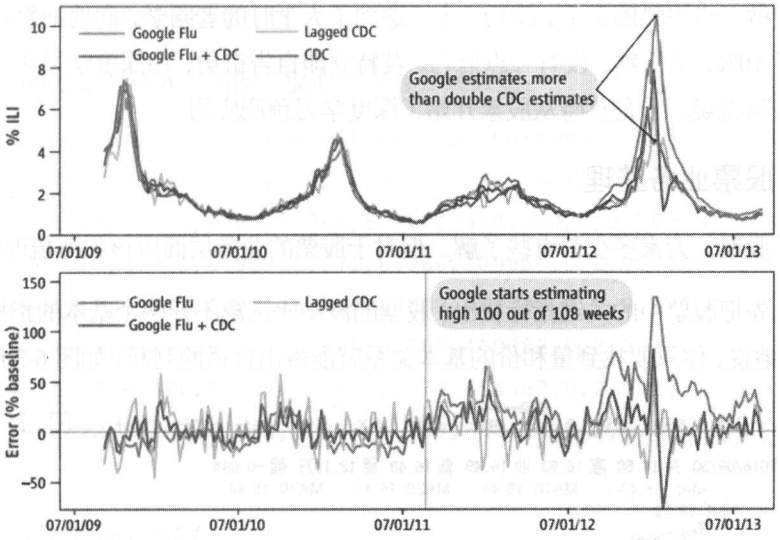


图 6-2 GFT 预测值与真实情况对比

GFT 为什么会在做了很久“好学生”后，成绩突然就不好呢？英特总结了以下三个原因。

（1）一个人搜索感冒，也不一定就是病了，也许就是随便搜索玩玩，也就是说这种搜索行为与感冒的相关度较小，且完全没有因果性。

（2）GFT 判断出的疾病趋势有一个滞后性，对某些传染病来说可能完全没用，等你预测出来后疾病早已大规模扩散了，即使不看 GFT 也知道流感来了！

（3）这种预测有很大的偏差，因为公众并不了解疾病和症状的对应情况。比如搜索感冒，但很可能得了非典，关键词和疾病的对应关系无法建立。

以上这几点直接导致了 GFT 成绩和实际值出现很大偏差。Google 的疾病预测的失败是否预示着对现实中任何预测问题都没有解决方案了呢？接下来我们跟随英特一起，来看看这个问题的答案。

6.2 股票预测（一）

英特经过这么多项目的锤炼已经对深度学习越来越有兴趣了，在其他领域也想用

深度学习做一点更酷的事情，凑巧，今天遇到了大学时的老同学，向他抱怨自己在股市又亏了 10%，天天跌，没有一点信心。英特立即自告奋勇，用深度学习来试试吧。老同学将信将疑。于是英特从股票开始了深度学习预测应用。

6.2.1 股票业务整理

对于股票，大家多少都有些了解。但对于股票的业务层面应该如何梳理呢？

我们先把股票中的关键指标找出。股票的波动永远离不开两个基本的指标：量和价，简单地说，你只要找到量和价的基本关系就能做出合适的判断（如图 6-3 所示）。

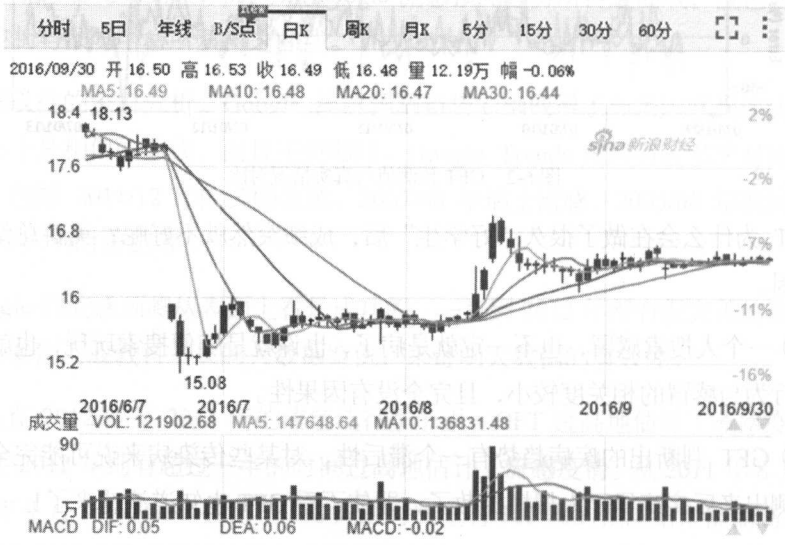


图 6-3 日线图

当然影响股票的不仅仅是量和价，像宏观经济、投资参考数据、分配预案、业绩预告和基金持股等均对股价有影响。宏观层面影响大盘，而具体到个股来说，相关的行业、概念、地域、盈利能力等都是个股的影响因素。更进一步来说投资人（股民）对于个股的信心有多少？这个信心如何反应出来？也就是说，投资人对股票的信心也是影响因素之一。

为了简单起见，我们把需要预测结果仅限定为上升或下降两类，而不考虑上升和下降的幅度。如要考虑，整体的预测精度将大大下降，性价比也不高。

第一步，先提取个股数据。

以000002（万科A）为例，提取并查看日线数据：

```
date,open,close,high,low,volume,code,turnover
2014-04-14,6.997,6.98,7.05,6.892,530246.0,2,0.55
2014-04-15,6.918,6.83,6.962,6.777,758813.0,2,0.78
2014-04-16,6.812,6.927,6.971,6.786,709656.0,2,0.73
2014-04-17,6.936,6.821,6.944,6.803,457592.0,2,0.47
2014-04-18,6.777,6.848,6.865,6.707,410082.0,2,0.42
2014-04-21,6.786,6.742,6.865,6.733,454483.0,2,0.47
2014-04-22,6.733,6.812,6.839,6.707,648158.0,2,0.67
……
```

股票基本数据如下。

```
code,name,industry,area,pe,outstanding,totals,totalAssets,liqu
idAssets,fixedAssets,reserved,reservedPerShare,esp,bvps,pb,timeToM
arket,undp,perundp,rev,profit,gpr,npr,holders
    300630,N普利,化学制药,海
南,28.96,0.31,1.22,43972.15,17459.27,9017.96,10207.42,0.84,0.571,3
.35,4.94,20170328,9244.87,0.76,22.28,37.62,76.75,28.14,55221
    603833,N欧派,家居用品,广
东,31.53,0.42,4.15,554842.75,219891.17,154291.38,65022.61,1.57,2.2
88,7.72,9.35,20170328,173348.88,4.18,27.23,94.39,36.55,13.31,39144
    603178,N圣龙,汽车配件,浙
江,26.79,0.5,2.0,152049.06,60084.26,55250.7,1732.39,0.09,0.405,2.7
2,3.98,20170328,21428.82,1.07,12.33,17.81,23.24,6.46,49497
    600540,新赛股份,种植业,新
疆,0.0,4.63,4.71,311359.59,156800.34,72435.15,80475.07,1.71,-0.012
,2.3,4.1,20040107,-23466.72,-0.5,46.09,72.79,6.71,-1.09,35548
……
```

这里的日线和股票基本数据使用tushare库获得。

GDP数据：

```
,quarter,gdp,gdp_yoy,pi,pi_yoy,si,si_yoy,ti,ti_yoy
0,2016.4,744127.2,6.7,63670.7,3.3,296236.0,6.1,384220.5,7.8
1,2016.3,532845.9,6.7,40665.7,3.5,209415.3,6.1,279890.2,7.6
2,2016.2,342316.4,6.7,22096.7,3.1,134250.4,6.1,184289.6,7.5
3,2016.1,161572.7,6.66,8803.0,2.95,61325.0,5.82,91444.7,7.6
4,2015.4,689052.1,6.9,60862.1,3.9,274277.8,6.0,346149.7,8.3
.....
```

货币供应量数据如下。

```
,month,m2,m2_yoy,m1,m1_yoy,m0,m0_yoy,cd,cd_yoy,qm,qm_yoy,ftd,f
td_yoy,sd,sd_yoy,rests,rests_yoy
0,2016.12,1550066.67,11.33,486557.24,21.35,68303.87,8.05,41825
3.37,--,1063509.43,--,307989.61,--,603504.20,--,152015.62,--
1,2016.11,1530432.06,11.39,475405.54,22.65,64903.50,7.58,41050
2.04,--,1055026.52,--,309067.58,--,597374.12,--,148584.82,--
2,2016.10,1519485.40,11.64,465446.65,23.85,64214.93,7.20,40123
1.71,--,1054038.76,--,307895.68,--,594169.86,--,151973.22,--
3,2016.9,1516360.50,11.50,454340.25,24.70,65068.62,6.60,389271
.63,--,1062020.26,--,315077.15,--,598880.53,--,148062.57,--
.....
```

英特的团队查看了 10 多个不同的类型的经济数据、股票数据，定下了基本的数据获取要求。

（1）股票的日线和 30 日线数据。

（2）舆论数据：Sina 论坛发帖数据。

（3）经济环境（宏观数据）：货币供应量、存款准备金、CPI。

现在需要预测的是第二天的股票涨跌，我们用[0,1]（涨），[1,0]（跌）来代表。

这里有些同学肯定要说了，你应该做关联分析啊，你应该做相关度分析啊，你起码做个方差啊，啥都没做你就确定用这些数据？其实这些方法都是从统计学上寻找关联。而经济学中的各个指标影响并不是直接的，我们要找到一条拟合的线或者在多维

空间上找到一个“曲面”是几乎不可能的（即使找到，代价也很大，无法计算）。我们的最终目的不是找到参数是否和最终股价波动有关联，而是直接预测股票。深度学习的优势就是在这些特征里变换、整合、选取，所以我们大致地圈好特征范围，相信大致的这些特征肯定对于股票是有影响的，之后的特征抽取就交给神经网络去做吧。神经网络训练好以后，可以观察训练后的模型的权重去除特征，虽然去不去特征影响都不会太大。如果非要做特征预抽取的话可以考虑用 PCA。

6.2.2 数据获取和准备

基于上一步的数据需求，获取以下基本数据。

```
def download_from_tushare(code):
    '''
    #宏观经济形势,数字指标 1:货币供应量
    month :统计时间
    m2 :货币和准货币（广义货币 M2）(亿元)
    '''
    import tushare as ts
    path = './data/'

    # 1 day line

ts.get_hist_data(str(code)).to_csv(path+"stock_data/"+str(code)+'.
csv')
    # 30 day lines
    ts.get_hist_data(str(code),
ktype='M').to_csv(path+"stock_data/"+str(code)+'_month.csv')

#获取货币供应量
    ts.get_money_supply().to_csv(path+'money_supply.csv')
    ts.get_gdp_quarter().to_csv(path+'gdp_quarter.csv')
ts.get_gdp_year().to_csv(path + 'gdp_year.csv')
#获取 cpi 数据
    ts.get_cpi().to_csv(path+'cpi.csv')
```

```
    ts.get_hist_data('sz').to_csv(path + 'sz.csv')
    #存款准备金率
ts.get_rrr().to_csv(path + 'rrr.csv')
```

这里将所有的数据以 csv 的格式存储起来，以便后续处理使用，比如经济数据和个股数据存储（如图 6-4 所示）。

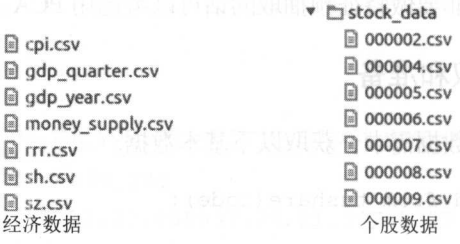


图 6-4　经济数据和个股数据

得到了数据后要根据前期数据来处理了，开始准备 X 训练集。

下面这段代码给出了如果要预测下一天的结果时，需要准备的 X。

```
def get_today_data_for_MLP(code):
    '''
    :param code:股票代码
    :return: 预测第二天，准备的 X
    '''
    import numpy as np
    data_path = "./data/stock_data/"
    oneDayLine, date = load_data_from_tushare(data_path +
str(code) + '.csv')
    volumn, volumn_dates = load_volumn_from_tushare(data_path +
str(code) + '.csv')
    daynum = 5
    X = []
    ef = Extract_Features()
    for i in range(daynum, len(date)):
        X_delta = [oneDayLine[k] - oneDayLine[k - 1] for k in range(i
- daynum, i)] + \
```

```
                [volumn[k] for k in range(i - daynum, i)] + \
                [float(ef.parse_weekday(date[i]))] + \
                [float(ef.lunar_month(date[i]))] + \
                [ef.rrr(date[i])] + \
                [ef.MoneySupply(date[i])]
        X.append(X_delta)

    X = preprocessing.MinMaxScaler().fit_transform(X)
    return np.array(X[-1])
```

我们来看下面这一段：

```
        X_delta = [oneDayLine[k] - oneDayLine[k - 1] for k in range(i
- daynum, i)] + \
                [volumn[k] for k in range(i - daynum, i)] + \
                [float(ef.parse_weekday(date[i]))] + \
                [float(ef.lunar_month(date[i]))] + \
                [ef.rrr(date[i])] + \
                [ef.MoneySupply(date[i])]
```

第一行 [oneDayLine[k] - oneDayLine[k - 1] for k in range(i - daynum, i)]是去股票的收盘价格，其中的 i 我们定为 5 天，5 天等于一周的交易日，这也是股票中周线的一个单位。这样我们就能保证周线特征和日线特征都加入进来。

还有，我们用第一天和第二天的差值来作为取值结果，能更好地表征波动情况。

第二行 [volumn[k] for k in range(i - daynum, i)]获取的是 5 个交易日的量能。

第三行 [float(ef.parse_weekday(date[i]))] 表示一周中的第几天。

第四行 [float(ef.lunar_month(date[i]))]获取的是农历月份。

第五行 [ef.rrr(date[i])] 获取的是存款准备金率，宏观经济。

第六行 [ef.MoneySupply(date[i])] 获取的是货币供应量，宏观经济。

这六行代码很重要，也是我们关键的特征处理步骤。

这里我们组装了一个X后，发现X的整体值类型太多，有[0,1]二值，有1347183.00反应量的超大值，如果直接放入模型计算的话，噪音就太多，有必要做归一化。什么叫归一化呢？简单来说就是把各个特征的尺度控制在相同的范围内，防止绝对值大的维度对结果的影响过大。这里我们选用最大最小值方法来做归一化。

```
X = preprocessing.MinMaxScaler().fit_transform(X)
```

没有做归一化前我们看到是这样的：

```
[12857372.0, 0.00656742556917682, 0.0021748586341888087,
-0.007378472222222142, 0.04984696108439005, -0.10883773799374823,
1.918048469387755, -0.2624849743197465, 0.018965772707067712,
2.888032572342591, 0.14, 0.4, 0.3, 0.3, 1.18, 0.003369330453563689,
0.20219938842567081, -0.00021143358187138505,
-0.0015351483717793908, 0.2791851260697089, 0.0028789303719332043,
0.098068448383464443, 0, 1, 1]
```

归一化后是这样的：

```
[ 0.55716091  0.53250846  0.51061976  0.46301435  0.74817568
0.02320841

  0.07733963  0.01910502  0.02662161  0.1032446   0.00489168
0.0230608

  0.01607268  0.01607268  0.07756813  0.51650956  0.20686533
0.57632332

  0.58488821  0.26139233  0.71495309  0.65249301  0.          1.
1.]
```

到这里，我们的数据准备就已经基本就绪。

接着再对训练集中的y值做处理，非常简单：

```
y_clf.append([1, 0] if oneDayLine[i] - oneDayLine[i - 1] >= 0 else
[0, 1])
```

同样需要做归一化：

```
y_clf = preprocessing.MinMaxScaler().fit_transform(y_clf)
```

另外要注意的是，神经网络算法对于训练数据数量是有要求的，这里英特取的是从2014年1月到目前为止，包含400条数据（有些股票中间有长时间停牌，数据不到400条）以上的那些股票，小于400条的股票不处理。

```
        if len(oneDayLine) < 400:
            continue
```

还要注意的是异常数据处理。举个例子，股票停牌了，复牌的当然大多数情况都是大涨大跌，如果你将数据连接起来，不考虑停牌情况，用这些数据训练模型将会产生偏差，也就是会给出错误的买入或卖出信号。

除了这些情况外，还要考虑节假日的影响、外盘的影响、国际环境的影响等。

6.2.3 模型搭建

英特团队根据股票的特点，并参考了一些论文，初步圈定了几种算法：MLP、LSTM、xgboost。

我们先来看看MLP的网络实现。

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation

def clf_model(input_dime):
    out = input_dime * 2 + 1
    model = Sequential()
    model.add(Dense(input_dim=input_dime, output_dim=out))
    model.add(Activation('tanh'))
    model.add(Dense(input_dim=out, output_dim=input_dime))
    model.add(Activation('tanh'))
    model.add(Dense(input_dim=input_dime, output_dim=2))
    model.add(Activation('sigmoid'))
    model.compile(optimizer='adam',
```

```
                loss='binary_crossentropy',
                metrics=['accuracy'])
    return model
```

该段代码英特实验了很久，先来看看，算法组一共用了 4 层（不算输入层，则一共 3 层）。

先看 out = input_dime * 2 + 1 这是对第一层隐层单元个数的确定，而这个对应个数英特定在了输入层的 2n+1 个。当然这里的单元个数可以更换，但为了让神经网络充分地寻找可靠的特征，一般要大于输入层个数。2n+1 只是一个经验值，读者可以根据实际情况进行调整。

看这一行 model.add(Activation('tanh'))，英特依照股票的特性选择隐层的激活函数'tanh'，这个激活函数提供[-1,1]分布范围。

Model 的最后一层激活函数选用了'sigmoid'，因为这里是一个分类模型，最后一句选用 Adam 是一种更好的随机优化方法。

```
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Adam 介绍[1]

Adaptive Moment Estimation(Adam) 是一种优化随机目标函数的算法。它和 Adadelta 以及 RMSprop 的区别在于计算历史梯度衰减方式不同，它不使用历史平方衰减，其衰减方式类似动量，如下：

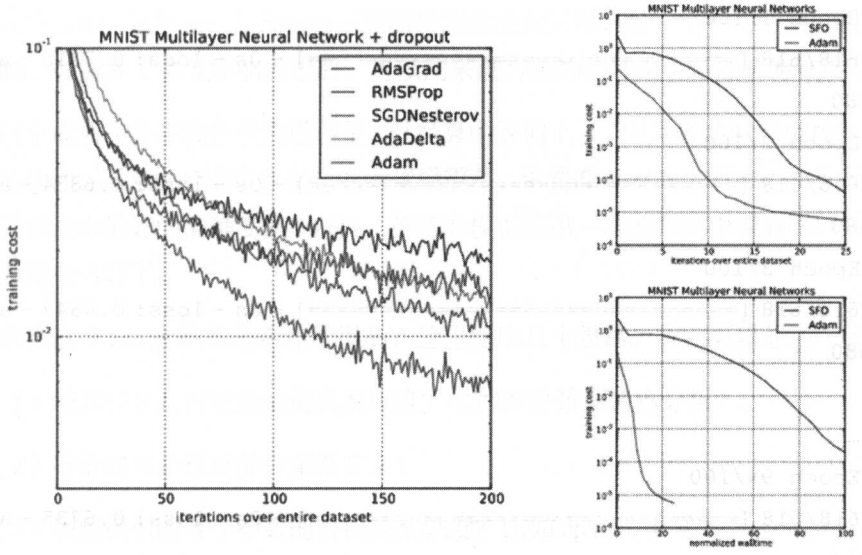
$$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t (\text{Update biased first moment estimate})$$

$$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_1) \cdot g_t^2 (\text{Update biased second raw moment estimate})$$

研究者们将 Adam 与其他的几个自适应学习速率进行了比较，效果均更好，见图 6-5。建议默认值：$\beta 1$=0.9，$\beta 2$=0.999，ϵ=10−8。这些参数在 Keras 中是默认值，无需修改。

---

1 参考自论文 https://arxiv.org/abs/1412.6980v8。



(a) (b)

图 6-5 Adam 和其他优化方法对比图

模型准备好后，我们将测试数据和训练数据分开：X_clf_train, X_clf_test, y_clf_train, y_clf_test = create_Xt_Yt(X_clf, y_clf, 0.85)#0.8。

这里我们选取所有股票数据中的 0.85 作为训练数据 ，剩下的 0.15 作为测试数据将 epoch 定为 100 ， batch_size 定为 50，开始训练。

```
model = clf_model(input_dime)
model.fit(X_clf_train,
          y_clf_train,
          nb_epoch=100,
          batch_size=50,
          verbose=1
          )
```

结果如下：

```
this is code  600082
download over ~
```

```
    Epoch 1/100
    618/618 [==============================] - 0s - loss: 0.7315 - acc:
0.5680
    Epoch 2/100
    618/618 [==============================] - 0s - loss: 0.6834 - acc:
0.5680
    Epoch 3/100
    618/618 [==============================] - 0s - loss: 0.6847 - acc:
0.5680

    ......
    Epoch 99/100
    618/618 [==============================] - 0s - loss: 0.6735 - acc:
0.5906
    Epoch 100/100
    618/618 [==============================] - 0s - loss: 0.6723 - acc:
0.5898
    Saved model to disk
     10/110 [=>............................] - ETA:
0s****************************************
    code = 600082
    ****************************************
    test :  loss 0.686093173244 acc 0.58181818236
    acc: 58.18%
```

600082 这支股票测试下来的 acc 为 58.18%，嗯，还不错！英特很高兴，看来这个模型能走得通。

6.2.4 优化

只有一支股票的准确率评估是没有意义的，现在就做下整体的评估。英特团队将股票代码从 600000 到 603990 所有股票数据全部下载下来，一一评测并取平均值，得到以下 acc。

```
    0.527285737208
```

众所周知，如果 acc 在 50%的话和随机测量没有区别，而一个平均结果已经超过了 50%，在摒弃了其他影响因素后，可以用来进行预测，理论上是能够赚钱的。

这个成绩在各种股票预测模型里，已经算中等偏上了，因为各个股票市场有其自有的波动性，不考虑特别事件引起的大规模震荡，单看准确率数字而言，目前所了解的公开算法中最好的准确度是 53.6%，英特的模型在第一次的预测中得到 52.7%，已经算做得相当好了。

那么这个 acc 准确率是否还能提高？这里给出几个思路，大家可以思考一下。

（1）增加特征是否能继续提高准确度？比如舆论数据情感分析？

（2）特征的初始权重能否重新定义？

（3）调整模型的各个参数能否提高准确度？比如 epoch、lr 等？

（4）调整特征的时间范围？

（5）对于不同行业的股票分开处理，增加不同的特征是否更好？比如我们给一个农业股票加入大葱、大蒜期货的平仓价格特征，或加入天气信息？

（6）就拿 MLP 来说，模型的结构能否改进？增加多隐层？

（7）现有的方式是用股票数据最后 15%这个部分来做验证，模型是否可以选取其他验证方式？既可以达到最高的数据利用率，又能更好的评价模型，比如交叉验证？

6.2.5 后续

要想把预测结果应用于真实的股票预测，只是得到股票第二日上涨的概率并不是完整的过程，只计算第二天上涨下跌的概率并评分也不是最终的目标，如何应用这些概率值来操纵股票的买卖，才是我们的目标。一起来看看英特是怎么做的。

第一步，根据模型的输出，将结果存下来，存储格式如下：

```
002007,0.7027220644035449
600733,0.70033955857385399
002709,0.70029673661251801
```

```
002098,0.69875776397515532
603169,0.69077567959231656
600298,0.68972895710002335
.....
```

第二步，要设定股票购买规则。购买规则就是要定清楚买入哪支股票，以及买入多少。比如说英特的购买规则：每个股票每次购买 20000 元，如果没到 10 手并且第二天仍旧预测的是买入则再购买 20000 元，直到达到 10 手为止，而卖出时机则按照概率值确定。

```
# 预测下一天会涨的概率大于 0.71，则买进约 20000 元
if (float(read_result[model_index][1]) >=Acc_threshold and
model_all[model_index].predict(X_all[model_index].reshape((1,
X_all[model_index].shape[0])))[0][0] > 0.71)
```

这里英特还想了个好办法以利用那些准确度特别低的股票，比如下面这些股票：

```
600589,0.35267702936096719
600708,0.35218068535825549
600120,0.34633204633204635

000979,0.3381355938264879
002355,0.33637901901593665
601288,0.33209700419457251
002417,0.32599653379549394
.....
```

如果预测准确度特别低，则按预测值做反向操作，比如 601766 acc 为 0.37，用模型预测出了明天的方向是下跌，那我们就按照反向操作，即进行买入操作。

```
# 预测下一天会涨的概率大于 0.71，则买进约 10000 元
if (float(read_result[model_index][1]) >=acc_threshold and
model_all[model_index].predict(X_all[model_index].reshape((1,
X_all[model_index].shape[0])))[0][0] > 0.71) or \
```

```
    (float(read_result[model_index][1]) <=acc_low_threshold and
model_all[model_index].predict(X_all[model_index].reshape((1,
X_all[model_index].shape[0])))[0][0] <= 0.4):
```

英特按照这个买卖规则，做了一个月的真实股票模拟测试，模拟测试结果每月有6%以上的盈利。

总结：我们在做评测时一定要定义清楚目标，模型的准确率很多时候只是一个中间步骤，从股票这个案例中看，目标是盈利，而模型往往只能给出准确率，从计算概率到最终的盈利的转换过程，就是你的学习业务的过程，最终所有的回报都来自最后的结果，而不是模型的概率。

思考一下，如果你是金融领域从业人员，需要将新股发行给合适的自然人和机构，那么目标即是最容易接受这只新股的自然人或者机构，换句话说也就是求自然人或机构的行为模式特征，这有点像一个推荐系统。

我们的思路是找到这个自然人或者机构原有操纵股票的行为模式，根据历史新股的购买记录来确定，哪一类自然人或者机构更有可能购买马上要发行的股票。这个模型的核心部分和上面我们提到的数据和算法比较类似，只是最后要求的 y 并不是上涨或下跌，而是买卖这只股票的行为特征，输入数据只是这个自然人或者机构的全部交易记录。摩根大通已在实际交易中使用了以上算法思路，如果你是金融从业者也可以试一下。

6.3 股票预测（二）

英特在完成股票预测后又接到一个设备故障预测的项目，该项目大致情况是这样的：主要目标是预测每台设备未来 7 天发生故障的可能性，每台设备有 200 个传感器，每个传感器平均产生 10～15 组数据，时间间隔是 1 秒，也就是 1 秒钟每台设备要产生 2000～3000 组数据。由于特征维度较多，英特首先做了一次 PCA[1] 特征筛选。筛选出 1000 维特征，并形成趋势线图，由专门负责监控维护的老师傅看了两个月的每秒

1 主成分分析。

1000 个数据的趋势线，人工总结出了 18 种出现异常前的趋势图形波动。然后将 18 种情况的特征（数值特征）放入模型中考虑，一旦发现其中一种，再结合其他特征，精确地预测出 7 天内故障发生情况。

做完项目，英特得到一个启发：为什么要人工总结出 18 种图形波动，让 CNN 网络来帮我“看”，看完后帮我总结出图形特征不是更好吗？而股票正好有非常符合用神经网络去“看”的图形，这就是 K 线图！

英特说干就干，先整理下思路。输出维度还是用涨跌定义，也就是一个[0,1]二值化向量（当然也可以用卷积网络输出 32 维的向量，参与到之前的经济、大盘等参变量中一起运算）。

输入就是一幅图像了，但是这图像怎么看？具体看多少天？英特据此咨询了自己的老同学，了解到一般 K 线图要个股结合大盘一起看，短线起码要看 5 日线、10 日线，所以英特就以 10 日线的 K 线图形作为输入数据，挑选那些第二天会涨的前 10 日的 K 线图。这样就确定好了数据的基本输入输出（如图 6-6 所示）。

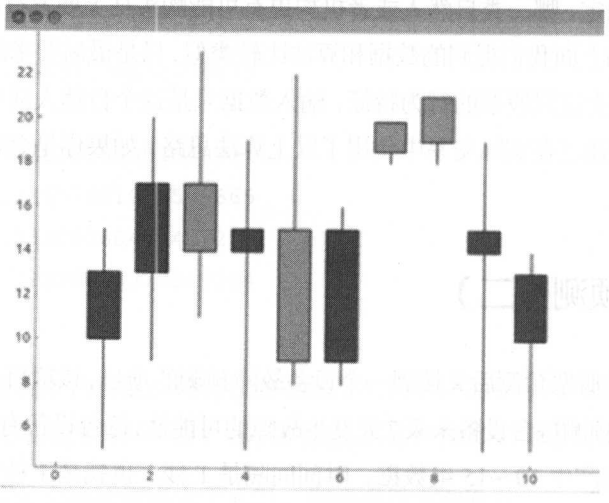


图 6-6　用数据制作的 K 线图

英特用自己写的多层卷积和经典的 VGG16 网络分别做了测试。

下面来看下简单的多层卷积代码。

```
model = Sequential()
```

```
    model.add(Convolution2D(64, 3, 3, init='glorot_uniform',
subsample=(1, 1), input_shape=(1, img_width, img_height)))
    model.add(advanced_activations.LeakyReLU(alpha=0.3))
    model.add(MaxPooling2D(pool_size=(2, 2), dim_ordering="th"))

    model.add(Convolution2D(128, 3, 3, subsample=(1, 1),
dim_ordering='th'))
    model.add(advanced_activations.LeakyReLU(alpha=0.3))
    model.add(AveragePooling2D(pool_size=(2, 2),
dim_ordering="th"))

    model.add(Convolution2D(128, 3, 3, subsample=(1, 1),
dim_ordering='th'))
    model.add(advanced_activations.LeakyReLU(alpha=0.3))
    model.add(AveragePooling2D(pool_size=(2, 2),
dim_ordering="th"))

    model.add(Convolution2D(64, 2, 2, subsample=(1, 1),
dim_ordering='th'))
    model.add(advanced_activations.LeakyReLU(alpha=0.3))
    model.add(AveragePooling2D(pool_size=(2, 2),
dim_ordering="th"))

    model.add(Dropout(0.5))
    model.add(Dense(8))
    model.add(advanced_activations.LeakyReLU(alpha=0.3))

    model.add(Dense(1))
    model.add(Activation('sigmoid'))

    model.compile(loss='binary_crossentropy',
                  optimizer='rmsprop',
                  metrics=['accuracy'])
```

英特又使用了VGG16网络来对比，结果发现VGG能在40左右的epoch就收敛到acc为0.98左右，而英特自己搭建的简单卷积由于层数的深度还是特征提取都无法满足这类图片的要求。

英特又对卷积网络做了一些简单的改进：

（1）卷积层提取特征做了扩充：从原来的64-128变成64-128-256卷积；

（2）增加了两层；

（3）优化方法使用了传统的SGD。

具体代码如下。

```
model.add(Convolution2D(64, 3, 3, init='glorot_uniform',
subsample=(1, 1), input_shape=(1, img_width, img_height)))
model.add(advanced_activations.LeakyReLU(alpha=0.3))
model.add(MaxPooling2D(pool_size=(2, 2), dim_ordering="th"))

model.add(Convolution2D(128, 3, 3, subsample=(1, 1),
dim_ordering='th'))
model.add(advanced_activations.LeakyReLU(alpha=0.3))
model.add(MaxPooling2D(pool_size=(2, 2), dim_ordering="th"))

model.add(Convolution2D(128, 3, 3, subsample=(1, 1),
dim_ordering='th'))
model.add(advanced_activations.LeakyReLU(alpha=0.3))
model.add(MaxPooling2D(pool_size=(2, 2), dim_ordering="th"))

model.add(Convolution2D(128, 3, 3, subsample=(1, 1),
dim_ordering='th'))
model.add(advanced_activations.LeakyReLU(alpha=0.3))
model.add(AveragePooling2D(pool_size=(2, 2),
dim_ordering="th"))
```

```
    model.add(Convolution2D(256, 2, 2, subsample=(1, 1),
dim_ordering='th'))
    model.add(advanced_activations.LeakyReLU(alpha=0.3))
    model.add(MaxPooling2D(pool_size=(2, 2), dim_ordering="th"))

    model.add(Convolution2D(256, 2, 2, subsample=(1, 1),
dim_ordering='th'))
    model.add(advanced_activations.LeakyReLU(alpha=0.3))
    model.add(MaxPooling2D(pool_size=(2, 2), dim_ordering="th"))

    model.add(Dense(1))
    model.add(Activation('sigmoid'))

    model.compile(loss='binary_crossentropy',
                  optimizer=optimizers.SGD(lr=1e-5, momentum=0.9),
                  # optimizer='rmsprop',
                  metrics=['accuracy'])
```

同样也在 40 个 epoch 后得到了 loss: 0.3291、acc: 0.9255 的好成绩，看来英特的简单卷积潜力很大!

英特随机选取一支股票来做实验，例如股票代码 600028。由于生成的股票 k 线图两两很相像，要用卷积网络将它们分开，实际上是一件比较困难的事情，我们使用最简单的 SGD 来做优化函数，将参数尽可能地设置低一些（这里可以用 fine-tune 的思想来微调网络，即固定住之前的层，只放开最后的 FC 层和 softmax 层进行训练微调）。

```
    sgd2 = optimizers.SGD(lr=1e-5, momentum=0.9)
```

25 epoch 后的结果：

```
    Epoch 25/25
    196/196 [==============================] - 14s - loss: 0.6732 -
acc: 0.5969 - val_loss: 0.6831 - val_acc: 0.6000
    600028 val_acc is 0.639999985695
```

```
acc_result 1
******************************
0.639999985695
******************************
```

测试 34 张图片结果不错：

```
Found 34 images belonging to 1 classes.
[[ 0.49383062  0.50616938]
 [ 0.47110081  0.52889919]
 [ 0.49120966  0.50879037]
。。。。。。
 [ 0.47680157  0.52319837]
 [ 0.51430994  0.48569009]
 [ 0.5386976   0.46130234]]
```

在股票预测过程中英特发现如果仅仅可以观察个股的图形模式是不行的，需要将大盘的走势考虑进来，这里有两种思路：

（1）将网络分成两路网络，每路基准网络（这里基准网络可以是任意图像分类网络，例如：VGG16 或 Inception-v3）只预测一个图片，在最后将两路网络输出的向量做 concat，再对合并的向量做几次卷积，甚至简单点直接做几次 FC（如图 6-7 所示）。

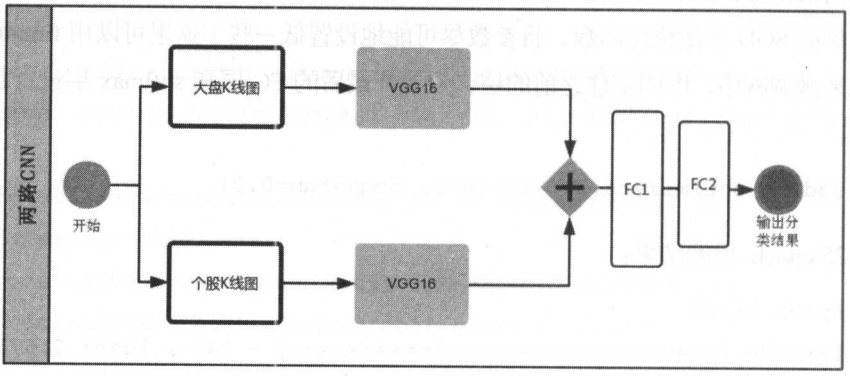


图 6-7　两路 CNN 算法结构图

（2）将个股 K 线图和大盘的 K 线图组合在一起输出，用一种基准 CNN 进行分类。英特这两种都实现了，但为了方便，选用了将两种 K 线图叠放的方式，叠放方式如图 6-8 所示，读者也可以想想哪种方式更适合 K 线图。

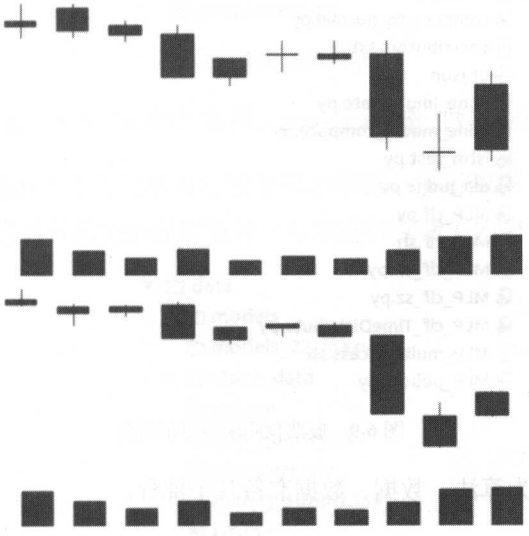


图 6-8 K 线图叠放

英特将股票预测第一部分的预测输出结果和这一部分的预测输出结果做了结合，输出最终预测分值，并用一个月的数据做了模拟测试，这次发现每月能盈利 8%。交付给老同学的时候，英特开玩笑说：“保证这次每月能赚 8 个点！”老同学笑笑，一副高深莫测的样子：“股市有风险，‘预测’需谨慎！”

（本章主要讨论深度学习在股票中的应用，请勿以此方法直接入市操作，否则后果自负。）

至此股票预测的部分英特已经开发完成，现在我们来做个简单的项目结构回顾一下（如图 6-9 所示）。

图 6-9　股票预测项目目录结构

英特将项目分为算法、数据、数据准备几个部分。

算法目录(algorithum)严格限定为只存储基础算法部分，例如 DNN、CNN、LSTM、RMSE[1]等算法（如图 6-10 所示）。

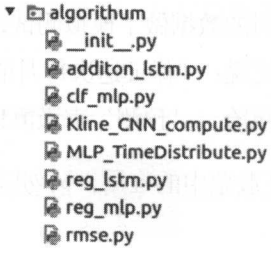


图 6-10　算法目录

数据准备（data_preprocess）目录有各种下载数据、处理数据的工具（如图 6-11 所示）。

1　均方根误差。

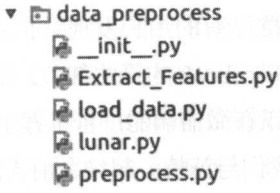


图 6-11 数据准备目录

比如 load_data.py 负责下载股票数据，preprocess.py 做数据准备。

而数据目录（data）下存储所有的股票原始数据、经济数据、舆论数据、模型、验证结果等等，并且对计算完的模型及评分分别进行存储（如图 6-12 所示）。

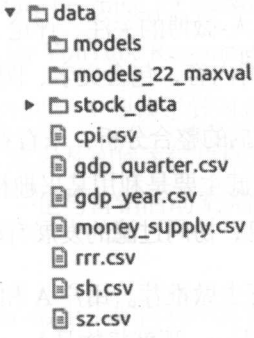


图 6-12 数据目录

此基本目录结构有比较清晰的分界线，适合大部分机器学习或深度学习应用开发。

6.4 深度学习在推荐领域的应用：Lookalike 算法

当 2012 年 Facebook 在广告领域开始应用定制化受众（Facebook Custom Audiences）功能后，受众发现这个概念真正得到大规模应用。什么是受众发现？如果你的企业已经积累了一定的客户，无论这些客户是否关注你或者是否和你在 Facebook 上有互动，你都能通过 Facebook 的广告系统触达到。受众发现实现了什么功能？在没有这个系统之前，广告投放一般情况都是用兴趣标签去区分用户，再去给这部分用户发送广告，受众发现让你不用选择这些标签，包括用户基本信息、兴趣等。你需要做的只是上传一批你目前已有的用户或者你感兴趣的一批用户，剩下的工作就等着受众功能帮你完成了。

像 Facebook 这样通过一群已有的用户发现并扩展出其他用户的推荐算法就叫 Lookalike，当然英特并不清楚 Facebook 的算法细节，而各个公司实现 Lookalike 的方式也各有不同。这里也包括腾讯在微信端的广告推荐上的应用、Google 在 YouTube 上推荐感兴趣视频等。下面让我们与英特一起结合前人的工作，实现自己的 Lookalike 算法，并尝试着在新浪微博上应用这一算法。

6.4.1 调研

首先要确定微博领域的数据，关于微博的数据可以采用下面这几个分类维度。

- 用户基础数据：用户的年龄、性别、公司、邮箱、地点、公司等。
- 关系图：根据人-人和人-微博的关注、评论、转发信息建立关系图。
- 内容数据：用户的微博内容，包含文字、图片、视频。

有了这些数据后，怎么做数据的整合分析？来看看现在应用最广的方式——协同过滤、或者叫关联推荐。协同过滤主要是利用某兴趣相投、拥有共同经验之群体的喜好来推荐用户可能感兴趣的信息，协同过滤的发展有以下三个阶段。

第一阶段：基于用户的喜好去做推荐。用户 A 和用户 B 相似，用户 B 购买了物品 a、b、c，用户 A 只购买了物品 a，那就将物品 b、c 推荐给用户 A。这就是基于用户的协同过滤，其重点是如何找到相似的用户。因为只有准确地找到相似的用户才能给出正确的推荐。而找到相似用户的方法，一般是根据用户的基本属性贴标签分类，再高级点可以用上用户的行为数据。

第二阶段：某些商品光从用户的属性标签找不到联系，而根据商品本身的内容联系倒是能发现很多有趣的推荐目标，它在某些场景中比基于相似用户的推荐原则更加有效。比如在购书或者电影类网站上，当你看一本书或电影时，推荐引擎会根据内容给你推荐相关的书籍或电影。

第三阶段：如果只把内容推荐单独应用在社交网络上，准确率会比较低，因为社交网络的关键特性还是社交关系。如何将社交关系与用户属性一起融入整个推荐系统就是关键。另外一个问题就是仅仅用兴趣标签过于粗犷，人与人的兴趣差异不光光是兴趣标签决定的，往往和时间、环境等其他的影响息息相关，如何将人在社交网络的所有特征尽可能提取出来并且计算呢？在神经网络和深度学习算法出现后，提取特征

任务就变得可以依靠机器完成了，人们只要把相应的数据准备好就可以了，其他数据都可以提取成向量形式，而社交关系作为一种图结构如何表示为深度学习可以接受的向量形式，而且这种结构还能有效还原原结构中位置信息？这就需要一种可靠的向量化社交关系的表示方法。基于这一思路，在 2016 年的论文中出现了一个算法 node2vec，使社交关系也可以很好地适应神经网络。这意味着深度学习在推荐领域应用的关键技术点已被解决。

在实现算法前英特主要参考了以下三篇论文：

- Audience Expansion for Online Social Network Advertising ,2016
- node2vec: Scalable Feature Learning for Networks　Aditya Grover ,2016
- Deep Neural Networks for YouTube Recommendations ,2016

第一篇论文是 Linkedin 给出的，主要谈了针对在线社交网络广告平台如何根据已有的受众特征做受众群扩展。这涉及如何定位目标受众和原始受众的相似属性。论文给出了两种方法来扩展受众，一是与营销活动无关的受众扩展，二是与营销活动有关的受众扩展。

在流程图 6-13 中，Linkedin 给出了如何利用营销活动数据、目标受众基础数据去预测目标用户行为进而发现新的用户。在今天的推荐系统或广告系统里越来越多地利用了多维度信息。如何将这些信息有效地加以利用，这篇论文给出了一条路径，而且在工程上这篇论文也论证得比较扎实，值得参考。

第二篇论文主要讲的是 node2vec，这也是本文用到的主要算法之一。node2vec 主要用于处理网络结构中的多分类和链路预测任务，具体来说是对网络中的节点和边的特征向量表示方法。

简单点来说就是将原有社交网络中的图结构，表达成特征向量矩阵，每一个 node（可以是人或物品或内容等）表示成一个特征向量，用向量与向量之间的矩阵运算来得到相互的关系。

下面来看看 node2vec 中的关键技术——随机游走算法，它定义了一种新的遍历网络中某个节点的邻域的方法，具体策略如图 6-14 所示。

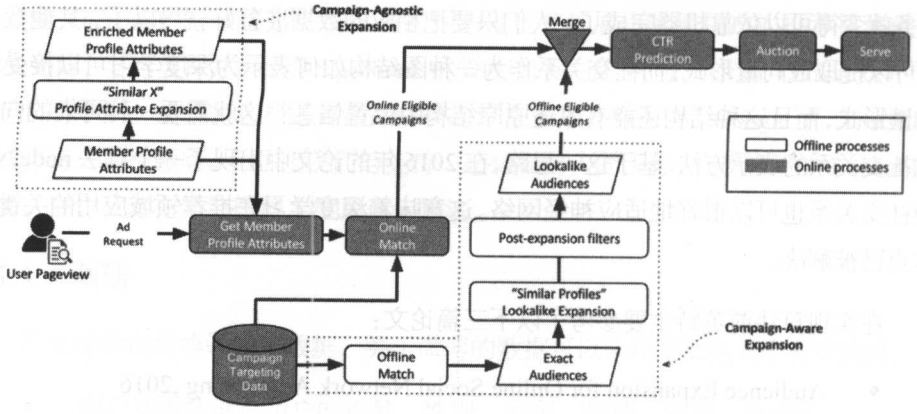


图 6-13　Linkedin 的 lookalike 流程图

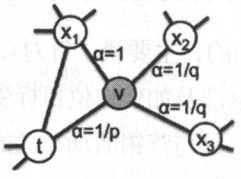


图 6-14　随机游走策略

假设我们刚刚从节点 t 走到节点 v，当前处于节点 v，现在要选择下一步该怎么走，方案如下：

$$\alpha_{pq}(t,x)=\begin{cases}1/p, & \text{if } d_{tx}=0\\ 1, & \text{if } d_{tx}=1\\ 1/q, & \text{if } d_{tx}=2\end{cases}$$

其中d_{tx}表示节点 t 到节点 x 之间的最短路径，$d_{tx}=0$表示会回到节点 t 本身，$d_{tx}=1$表示节点 t 和节点 x 直接相连，但是在上一步却选择了节点 v，$d_{tx}=2$表示节点 t 不与 x 直接相连，但节点 v 与 x 直接相连。其中 p 和 q 为模型中的参数，形成一个不均匀的概率分布，最终得到随机游走的路径。与传统的图结构搜索方法（如 BFS 和 DFS）相比，这里提出的随机游走算法具有更高的效率，因为本质上相当于对当前节点的邻域节点的采样，同时保留了该节点在网络中的位置信息。

node2vec 由斯坦福大学提出，并有开源代码[1]，这一部分大家不用自己动手实现

1　https://github.com/aditya-grover/node2vec。

了（本书的方法需要在源码的基础上改动图结构）。

第三篇论文讲的是 Google 如何做 YouTube 视频推荐，论文是在英特做完结构设计和流程设计后看到的，其中模型架构的思想和英特的不谋而合，还解释了为什么要引入 DNN：引入 DNN 的好处在于大多数类型的连续特征和离散特征可以直接添加到模型当中。此外英特还参考了这篇论文对于隐含层（FC）单元个数的选择。图 6-15 是这篇论文提到的算法结构。

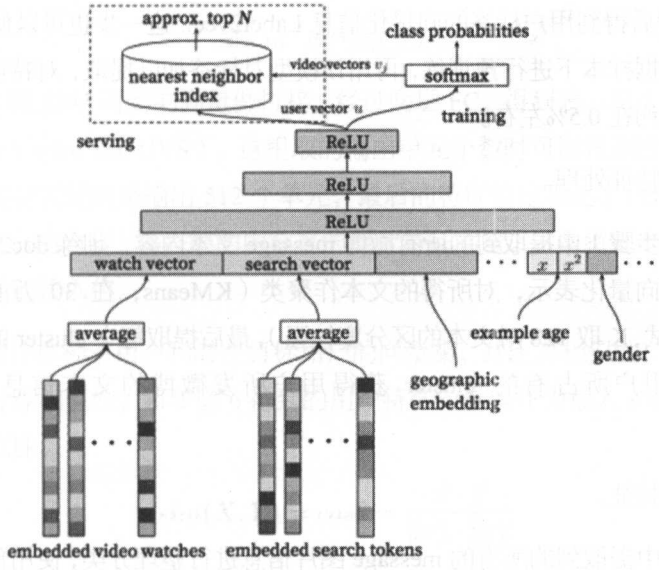


图 6-15 YouTube 推荐算法结构图

6.4.2 实现

（1）数据准备。

① 获得用户的属性（User Profile），如性别、年龄、学历、职业、地域、能力标签等；

② 根据项目内容和活动内容制定一套受众标签（Audience Label）；

③ 提取用户之间的关注关系，微博之间的转发关系；

④ 获取微博 message 中的文本内容；

⑤ 获得微博 message 中的图片内容。

（2）用户标签特征处理。

① 根据步骤 1 中用户属性信息和已有的部分受众标签系统。利用 GBDT 算法（可以直接用 xgboost）将没有标签的受众全部打上标签。这个分类问题中请注意处理连续值变量以及归一化。

② 将标签进行向量化处理，这个问题转化成对中文单词进行向量化，这里用 word2vec 处理后得到用户标签的向量化信息 Label2vec。这一步也可以使用 word2vec 在中文的大数据样本下进行预训练，再用该模型对标签加以提取，对特征的提取有一定的提高，大约在 0.5%左右。

（3）文本特征处理。

清洗整理步骤 1 中提取到的所有微博 message 文本内容，训练 doc2vec 模型，得到单个文本的向量化表示，对所得的文本作聚类（KMeans，在 30 万的微博用户的 message 上测试，K 取 128 对文本的区分度较强），最后提取每个 cluster 的中心向量，并根据每个用户所占有的 cluster 获得用户所发微博的文本信息的向量表示 Content2vec。

（4）图像特征。

将步骤 1 中提取到的所有的 message 图片信息进行整理分类，使用预训练卷积网络模型（这里为了平衡效率选取 VGG16 作为卷积网络）提取图像信息，对每个用户 message 中的图片做向量化处理，形成 Image2vec，如果有多张图片则将多张图片分别提取特征值再接一层 Max Pooling 提取重要信息后输出。

（5）社交关系建立（node2vec 向量化）。

将步骤 1 数据准备中获得的用户之间的关系和微博之间的转发评论关系转化成图结构，并提取用户关系 sub-graph，最后使用 node2vec 算法得到每个用户的社交网络图向量化表示。图 6-16 为社交关系化后的部分图示。

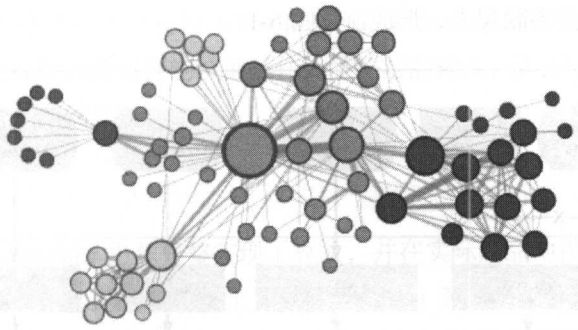


图 6-16 用户的社交关系

（6）将步骤 2345 得到的向量做拼接，经过两层 FC，得到表示每个用户的多特征向量集（User Vector Set, UVS）。这里取的输出单元个数时可以根据性能和准确度做平衡，目前英特实现的是输出 512 个单元，最后的特征输出表达了用户的社交关系、用户属性、发出的内容、感兴趣的内容等的混合特征向量，这些特征向量将作为下一步比对相似性的输入值。

（7）分别计算种子用户和潜在目标用户的向量集，并比对相似性。英特使用的是余弦相似度计算相似性，将步骤 6 得到的用户特征向量集作为输入 x 和 y，代入下面公式计算相似性。

$$sim(X,Y) = cos\theta = \frac{\vec{x} \cdot \vec{y}}{\|x\| \cdot \|y\|}$$

注意：余弦相似度更多是从方向上区分差异，而对绝对的数值不敏感，因此没法衡量每个维度值的差异。这里要在每个维度上减去一个均值或者乘以一个系数，或者在之前做好归一化。

（8）受众扩展。

① 获取种子受众名单，以及目标受众的数量 N；

② 检查种子用户是否存在于 UVS 中，将存在的用户向量化；

③ 计算受众名单中用户和 UVS 中用户的相似度，提取最相似的前 N 个用户作为目标受众。

最后将以上步骤串联起来，形成流程图 6-17。

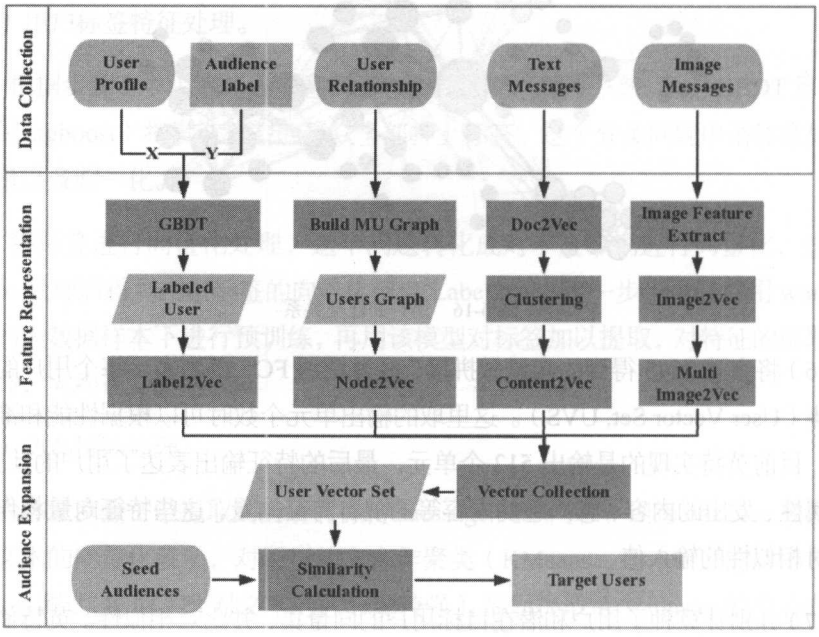


图 6-17　Lookalike 算法流程图

在以上步骤提取完特征后，英特使用一个两层的神经网络做最后的特征归并提取，算法结构示意图如 6-18 所示。

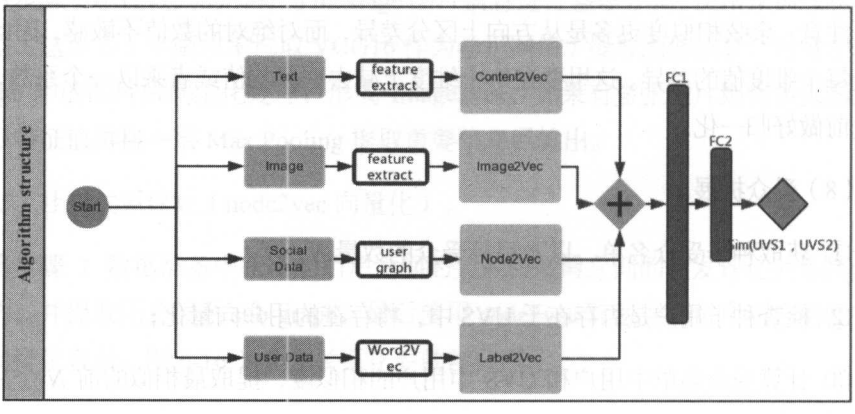


图 6-18　Lookalike 算法结构示意图

其中 FC1 层也可以替换成 Max Pooling，Max Pooling 层具有强解释性，也就是把用户特征群上提取的最重要的特征点作为下一层的输入，读者可以自行尝试，这里限于篇幅问题就不做展开了。

6.4.3 结果

英特根据 Lookalike 思想完整实现了算法，并在实际产品中投入试用，针对某客户（乳品领域世界排名前三的品牌主）计算出结果（部分）如下。

http://weibo.com/u/2800343060/home	苏州遇见烘焙工作室
http://weibo.com/u/5936697053/home	烘你欢心烘焙培训
http://weibo.com/u/5676210405/home	马佐烘焙西点培训
http://weibo.com/u/3227012590/home	流年 InCakeDesign
http://weibo.com/u/5699041328/home	韩式花朵蛋糕 Class

可以观察到以上微博 ID 的主题基本都是西点企业或西点培训企业，和品牌主售卖的乳品有很高的关联性：乳品是非常重要的西点原料，除终端用户外，西点相关企业就是乳品企业主需要寻找的最重要的受众之一。

6.4.4 总结探讨

英特在这个案例中学到了关于用深度学习做推荐的相关知识，在最终的总结会中，英特也列出了两点通过本项目学习到的经验。

（1）特征表达。

除了前文提到的特征外，英特也对其他重要特征表达做了处理和变换：根据业务需求，需要抽取出人的兴趣特征，如何表达一个人的兴趣？除了他自己生成的有关内容外，还有比较关键的一点是比如“我”看了一些微博，但并没有转发他们，大多数情况下都不会转发，但有些“我”转发了，有些“我”评论了；“我”转发了哪些？评论了哪些？这次距上次的浏览该人的列表时间间隔多久？都代表“我”对微博的兴趣，而间接地反映出“我”的兴趣特征。这些数据看来非常重要，但又无法直接取得，怎么办？

下面来定义一个场景，试图描述出我们对看过的内容中哪些是感兴趣的，哪些是不感兴趣的：

a）用户 A，以及用户 A 关注的用户 B；

b）用户 A 的每天动作时间（比如转发、评论、收藏、点赞的时间点）。其中起始时间定义为苏醒时间 A_wake(t)；

c）用户 B 每天发帖（转发、评论）时间：B_action(t)；

d）简单假设一下 A_wake(t)> B_action(t)，也就是 B_action(t)的评论都能看到。这就能得到用户 A 对应了哪些帖子；

e）同理，也可知用户 A 在 A_wake(t)时间内转发、评论了哪些帖子；

f）结合上次浏览间隔时间，可以描述用户 A 对哪些微博感兴趣（postive），哪些不感兴趣（negative）；

（2）全连接层的激活单元比对提升。

英特在学习 Google 那篇论文后，比对隐含层（也就是结构图中的 FC 层）各种单元组合产生的结果（如表 6-1 所示）。

表 6-1 YouTube 推荐模型隐含层单元选择对比

Hidden layers	weighted, per-user loss
None	41.6%
256 ReLU	36.9%
512 ReLU	36.7%
1024 ReLU	35.8%
512 ReLU → 256 ReLU	35.2%
1024 ReLU → 512 ReLU	34.7%
1024 ReLU → 512 ReLU → 256 ReLU	34.6%

Table 1: Effects of wider and deeper hidden ReLU layers on watch time-weighted pairwise loss computed on next-day holdout data.

Google 选择的是最后一种组合，英特在初期选用了 512 tanh→256 tanh 这种两层组合，后认为输入特征维度过大，512 个单元无法完整地表达特征，故又对比了 1024→512 组合，发现效果确实有微小提升，大概提升了 0.7%。另外模型的 FC 层输入在［-1,1］区间，考虑到 relu 函数的特点[1]故没有使用它，而是使用 elu 激活函数。测试效果比 tanh 函数提升了 0.3%～0.5%。

1 relu 本质上就是取最大值函数，并不是全区间可导的。

参考文献

[1] IBM Corp，IBM SPSS Modeler CRISP-DM Guide（2011）.

[2] Bater Makhabel，Learning Data Mining with R（2015）.

[3] 维基百科 https://www.wikipedia.org/.

[4] Hownet 知网，NTUSD 台湾大学，同义词词林，哈工大信息检索研究室.

[5] IMPLEMENTING A CNN FOR TEXT CLASSIFICATION IN TENSORFLOW http://www.wildml.com/2015/12/implementing-a-cnn-for-text-classification-in-tensorflow/.

[6] Yoon Kim，Convolutional Neural Networks for Sentence Classification （2014）.

[7] 《知识就是力量》机器人的智能，足够当好你的心理医生吗？http://news.xinhuanet.com/science/2016-03/01/c_135145105.htm.

[8] Oriol Vinyals，A Neural Conversational Model（2015）.

[9] Adam Geitgey，Machine Learning is Fun! Part 4: Modern Face Recognition with Deep Learning（2016）.

[10] 刘昕 CNN 的近期进展与实用技巧 https://zhuanlan.zhihu.com/p/21432547(2016).

[11] Christian Szegedy， Going Deeper with Convolutions（2014）.

[12] Whiteinblue，深度学习研究理解 11：Going deeper with convolutions http://blog.csdn.net/whiteinblue/article/details/43635575.

[13] Christian Szegedy，Rethinking the Inception Architecture for Computer Vision（2015）.

[14] Traphix， http://www.jianshu.com/p/0cc42b8e6d25（2016）.

[15] Kaiming，He，Deep Residual Learning for Image Recognition（2015）.

[16] Song Han，DSD: DENSE-SPARSE-DENSE TRAINING FOR DEEP（2017）.

[17] R-CNN (CVPR2014, TPAMI2015) Region-based Convolution Networks for Accurate Object detection and Segmentation.

[18] Shaoqing Ren, Kaiming He, Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks（2016）.

[19] Wei Liu, SSD: Single Shot MultiBox Detector（2015）.

[20] YOLO9000: Better, Faster, Stronger Joseph Redmon*†, Ali Farhadi*† 2016.12.25.

[21] Leon A. Gatys, A Neural Algorithm of Artistic Stylec（2015）.

[22] Andrej Karpathy，Li Fei-Fei，Deep Visual-Semantic Alignments for Generating Image Descriptions（2014）.

[23] Hrayr Harutyunyan， Hrant Khachatrian，Combining CNN and RNN for spoken language identification，https://yerevann.github.io/2016/06/26/combining-cnn-and-rnn-for-spoken-language-identification/.

[24] Akira Fukui etc., Multimodal Compact Bilinear Pooling for Visual Question Answering and Visual Grounding （2016）.

[25] Andrej Karpathy, Deep Reinforcement Learning: Pong from Pixels http://karpathy.github.io/2016/05/31/rl/（2016）.

[26] Volodymyr Mnih etc.，Playing Atari with Deep Reinforcement Learning（2013）.

[27] Adam Santoro,One-shot Learning with Memory-Augmented Neural Networks（2016）.

[28] Graves，A.，Wayne，G.，& Danihelka，I.，Neural turing machines. arXiv preprint arXiv:1410.5401.(2014).

[29] Graves，A.，Adaptive Computation Time for Recurrent Neural Networks. arXiv preprint arXiv:1603.08983.(2016).

Broadview® 博文视点·IT出版旗舰品牌
www.broadview.com.cn
技术凝聚实力·专业创新出版

深度学习算法实践

《深度学习算法实践》以一位软件工程师在工作中遇到的问题为主线，阐述了如何从软件工程的思维向算法思维转变，以及深度学习算法的概念与实践：比如在哪些场景下需要运用深度学习算法、如何将深度学习算法应用到任务中、提高工作效率？不仅如此，作者还结合程序员在工作中经常面临的产品需求，详细阐述了应该怎样从算法的角度来看待、分解需求，并结合经典的任务对深度学习算法做了清晰的分析：如何用RNN和CNN结合来提取深度文本特征？如何开始写一个Chatbot？如何在Chatbot中应用深度学习？强化学习为什么这么强大，它是万能的吗？强化学习可以用在什么地方？对于图形领域的深度网络来说，是否有通用的提高模型精度的方法？如何利用深度学习来预测股票的趋势？YouTube是如何推荐影片的，我们如何将YouTube的深度学习经验应用在推荐系统中……这些经典的应用案例，能让有志于学习深度学习的读者，快速地理解核心所在，并顺利地上手实践。

作者简介：

吴岸城

致力于深度学习在文本、图像领域的应用。曾在中兴通讯、亚信联创担任研发经理、技术经理等职务，现任菱歌科技首席算法科学家一职。

注册博文视点社区（www.broadview.com.cn）用户，即享受以下服务：

| 提勘误赚积分：可在【提交勘误】处提交对内容的修改意见，若被采纳将获赠博文视点社区积分（可用来抵扣购买电子书的相应金额）。

| 交流学习：在页面下方【读者评论】处留下您的疑问或观点，与作者和其他读者共同交流。

页面入口：http://www.broadview.com.cn/31793

上架建议：人工智能>深度学习

ISBN 978-7-121-31793-4

9 787121 317934 >

定价：79.00元

策划编辑：刘　皎
责任编辑：郑柳洁
封面设计：吴海燕

欢迎投稿
邮箱：Ljiao@phei.com.cn
电话：010-88254395
新浪微博：@皎丫子